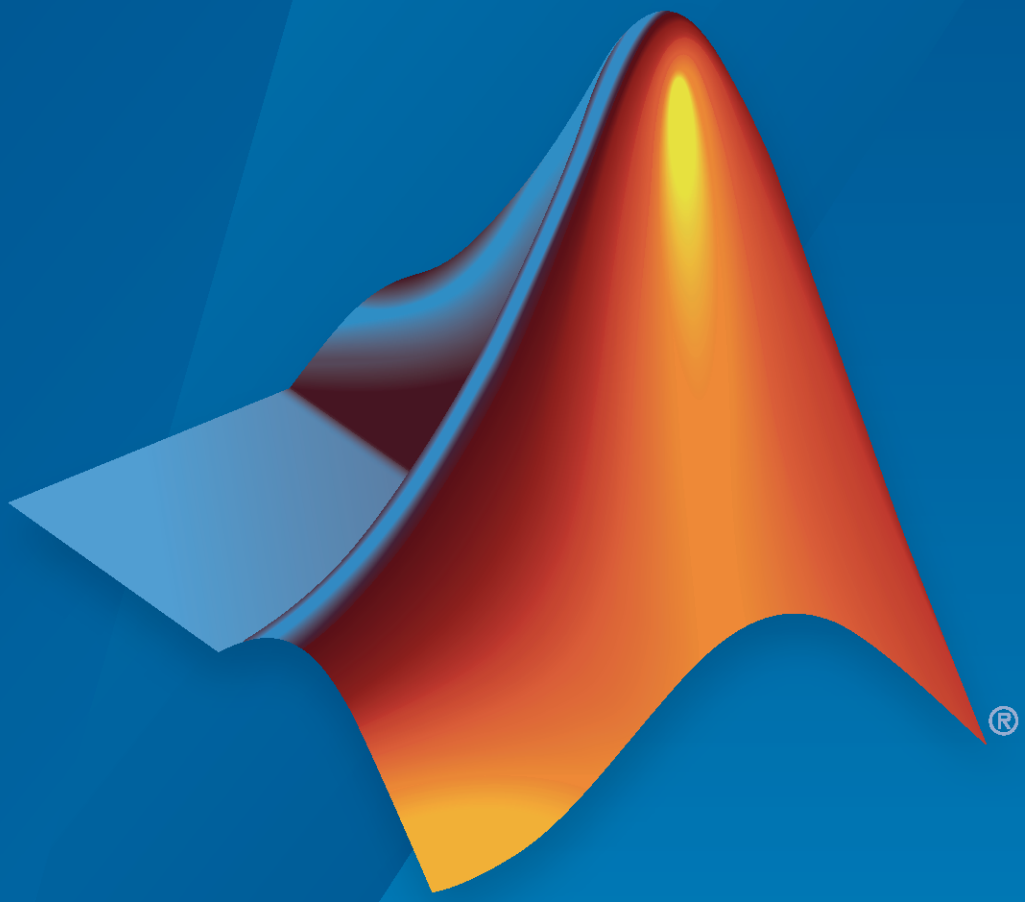
Polyspace® Bug Finder™

Reference



R2020a

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Polyspace® Bug Finder™ Reference

Revision History

September 2013	Online Only	New for Version 1.0 (Release 2013b)
March 2014	Online Only	Revised for Version 1.1 (Release 2014a)
October 2014	Online Only	Revised for Version 1.2 (Release 2014b)
March 2015	Online Only	Revised for Version 1.3 (Release 2015a)
September 2015	Online Only	Revised for Version 2.0 (Release 2015b)
October 2015	Online Only	Rereleased for Version 1.3.1 (Release 2015aSP1)
March 2016	Online Only	Revised for Version 2.1 (Release 2016a)
September 2016	Online Only	Revised for Version 2.2 (Release 2016b)
March 2017	Online Only	Revised for Version 2.3 (Release 2017a)
September 2017	Online Only	Revised for Version 2.4 (Release 2017b)
March 2018	Online Only	Revised for Version 2.5 (Release 2018a)
September 2018	Online Only	Revised for Version 2.6 (Release 2018b)
March 2019	Online Only	Revised for Version 3.0 (Release 2019a)
September 2019	Online Only	Revised for Version 3.1 (Release 2019b)
March 2020	Online Only	Revised for Version 3.2 (Release 2020a)

Contents

15 Approximations Used During Bug Finder Analysis

Analysis Options

Source code language (-lang)

Specify language of source files

Description

Specify the language of your source files. Before specifying other configuration options, choose this option because other options change depending on your language selection.

If you add files during project setup, the language selection can change from the default.

Files Added	Source Code Language
Only files with extension `.c`	`C`
Only files with extension `.cpp` or `.cc`	`CPP`
Files with extension `.c`, `.cpp`, and `.cc`	`C-CPP`

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See “Dependencies” on page 1-2 for ways in which the source code language can be automatically determined.

Command line: Use the option `-lang`. See “Command-Line Information” on page 1-3.

Settings

Default: Based on file extensions.

`C`

If your project contains only C files, choose this setting. This value restricts the verification to C language conventions. All files are interpreted as C files, regardless of their file extension.

`CPP`

If your project contains only C++ files, choose this setting. This value restricts the verification to C++ language conventions. All files are interpreted as C++ files, regardless of their file extension.

`C-CPP`

If your project contains C and C++ source files, choose this setting. This value allows for C and C ++ language conventions. `.c` files are interpreted as C files. Other file extensions are interpreted as C++ files.

Dependencies

- The language option allows and disallows many options and option values. Some options change depending on your language selection. For more information, see the individual analysis option pages.
- If you create a Polyspace project or options file from your build system using the `polyspace-configure` command or `polyspaceConfigure` function, the value of this option is determined by the file extensions.

For a project with both `.c` and `.cpp` files, the language option `C-CPP` is used. In the subsequent analysis, each file is compiled based on the language standard determined by the file extensions.

Command-Line Information

Parameter: `-lang`
Value: `c | cpp| c-cpp`
Default: Based on file extensions
Example (Bug Finder): `polyspace-bug-finder -lang c-cpp -sources "file1.c,file2.cpp"`
Example (Code Prover): `polyspace-code-prover -lang cpp -sources "file1.cpp,file2.cpp"`
Example (Bug Finder): `polyspace-bug-finder -lang c -sources "file1.c,file2.c"`
Example (Code Prover): `polyspace-code-prover -lang c -sources "file1.c,file2.c"`
Example (Bug Finder Server): `polyspace-bug-finder-server -lang c -sources "file1.c,file2.c"`
Example (Code Prover Server): `polyspace-code-prover-server -lang c -sources "file1.c,file2.c"`

See Also

`C standard version (-c-version)` | `C++ standard version (-cpp-version)`

C standard version (`-c-version`)

Specify C language standard followed in source code

Description

Specify the C language standard that you follow in your source code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See “Dependencies” on page 1-5 for other options that you must enable.

Command line: Use the option `-c-version`. See “Command-Line Information” on page 1-5.

Why Use This Option

Use this option so that Polyspace can allow features specific to a C standard version during compilation. For instance, if you compile with GCC using the flag `-ansi` or `-std=c90`, specify `c90` for this option. If you are not sure of the language standard, specify `defined-by-compiler`.

For instance, suppose you use the boolean data type `_Bool` in your code. This type is defined in the C99 standard but unknown in prior standards such as C90. If the Polyspace compilation follows the C90 standard, you can see compilation errors.

Some MISRA C® rules are different based on whether you use the C90 or C99 standard. For instance, MISRA C C:2012 Rule 5.2 requires that identifiers in the same scope and name space shall be distinct. If you use the C90 standard, different identifiers that have the same first 31 characters violate this rule. If you use the C99 standard, the number of characters increase to 63.

Settings

Default: `defined-by-compiler`

`defined-by-compiler`

The analysis uses a standard based on your specification for `Compiler (-compiler)`.

See “C/C++ Language Standard Used in Polyspace Analysis”.

`c90`

The analysis uses the C90 Standard (ISO®/IEC 9899:1990).

`c99`

The analysis uses the C99 Standard (ISO/IEC 9899:1999).

`c11`

The analysis uses the C11 Standard (ISO/IEC 9899:2011).

Dependencies

- This option is available only if you set `Source code language (-lang)` to `C` or `C-CPP`.
- If you create a project or options file from your build system using the `polyspace-configure` command or `polyspaceConfigure` function, the value of this option is automatically determined from your build system.

  If the build system uses different standards for different files, the subsequent Polyspace analysis can emulate your build system and use different standards for compiling those files. If you open such a project in the Polyspace user interface, the option value is shown as `defined-by-compiler`. However, instead of one standard, Polyspace uses the hidden option `-options-for-sources` to associate different standards with different files.

Command-Line Information

Parameter: `-c-version`
Value: `defined-by-compiler` | `c90` | `c99` | `c11`
Default: `defined-by-compiler`
Example (Bug Finder): `polyspace-bug-finder -lang c -sources "file1.c,file2.c" -c-version c90`
Example (Code Prover): `polyspace-code-prover -lang c -sources "file1.c,file2.c" -c-version c90`
Example (Bug Finder Server): `polyspace-bug-finder-server -lang c -sources "file1.c,file2.c" -c-version c90`
Example (Code Prover Server): `polyspace-code-prover-server -lang c -sources "file1.c,file2.c" -c-version c90`

See Also

`C++ standard version (-cpp-version)` | `Source code language (-lang)`

Topics

“Specify Polyspace Analysis Options”
“C/C++ Language Standard Used in Polyspace Analysis”
“C11 Language Elements Supported in Polyspace”

C++ standard version (-cpp-version)

Specify C++ language standard followed in source code

Description

Specify the C++ language standard that you follow in your source code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See “Dependencies” on page 1-7 for other options that you must enable.

Command line: Use the option `-cpp-version`. See “Command-Line Information” on page 1-7.

Why Use This Option

Use this option so that Polyspace can allow features from a specific version of the C++ language standard during compilation. For instance, if you compile with GCC using the flag `-std=c++11` or `-std=gnu++11`, specify `cpp11` for this option. If you are not sure of the language standard, specify `defined-by-compiler`.

For instance, suppose you use range-based `for` loops. This type of `for` loop is defined in the C++11 standard but unrecognized in prior standards such as C++03. If the Polyspace compilation uses the C++03 standard, you can see compilation errors.

To check if your compiler allows features specific to a standard, compile code with macros specific to the standard using compiler settings that you typically use. For instance, to check for C++11-specific features, compile this code. The code contains a C++11-specific keyword `nullptr`. If the macro `__cplusplus` is not `201103L` (indicating C++11), this keyword is used and causes a compilation error.

```
#if defined(__cplusplus) && __cplusplus >= 201103L
    /* C++11 compiler */
#else
    void* ptr = nullptr;
#endif
```

If the code compiles, use `cpp11` for this option.

Settings

Default: `defined-by-compiler`

`defined-by-compiler`

The analysis uses a standard based on your specification for `Compiler (-compiler)`.

See “C/C++ Language Standard Used in Polyspace Analysis”.

`cpp03`

The analysis uses the C++03 Standard (ISO/IEC 14882:2003).

`cpp11`

The analysis uses the C++11 Standard (ISO/IEC 14882:2011).

`cpp14`

The analysis uses the C++14 Standard (ISO/IEC 14882:2014).

Dependencies

- This option is available only if you set `Source code language (-lang)` to `CPP` or `C-CPP`.
- If you create a project or options file from your build system using the `polyspace-configure` command or `polyspaceConfigure` function, the value of this option is automatically determined from your build system.

  If the build system uses different standards for different files, the subsequent Polyspace analysis can emulate your build system and use different standards for compiling those files. If you open such a project in the Polyspace user interface, the option value is shown as `defined-by-compiler`. However, instead of one standard, Polyspace uses multiple standards for compiling the files. The analysis uses the hidden option `-options-for-sources` to associate different standards with different files.

Command-Line Information

Parameter: `-cpp-version`
Value: `defined-by-compiler` | `cpp03` | `cpp11` | `cpp14`
Default: `defined-by-compiler`
Example (Bug Finder): `polyspace-bug-finder -lang c -sources "file1.c,file2.c" -cpp-version cpp11`
Example (Code Prover): `polyspace-code-prover -lang c -sources "file1.c,file2.c" -cpp-version cpp11`
Example (Bug Finder Server): `polyspace-bug-finder-server -lang c -sources "file1.c,file2.c" -cpp-version cpp11`
Example (Code Prover Server): `polyspace-code-prover-server -lang c -sources "file1.c,file2.c" -cpp-version cpp11`

See Also

`C standard version (-c-version)` | `Source code language (-lang)`

Topics

“Specify Polyspace Analysis Options”
“C/C++ Language Standard Used in Polyspace Analysis”
“C++11 Language Elements Supported in Polyspace”
“C++14 Language Elements Supported in Polyspace”

Compiler (`-compiler`)

Specify the compiler that you use to build your source code

Description

Specify the compiler that you use to build your source code.

Polyspace fully supports the most common compilers used to develop embedded applications. See the list below. For these compilers, you can run analysis simply by specifying your compiler and target processor. For other compilers, specify `generic` as compiler name. If you face compilation errors, explicitly define compiler-specific extensions to work around the errors.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-compiler`. See “Command-Line Information” on page 1-14.

Why Use This Option

Polyspace uses this information to interpret syntax that is not part of the C/C++ Standard, but comes from language extensions.

For example, the option allows additional language keywords, such as `sfr`, `sbit`, and `bit`. If you do not specify your compiler, these additional keywords can cause compilation errors during Polyspace analysis.

Polyspace does not actually invoke your compiler for compilation. In particular:

- You cannot specify compiler flags directly in the Polyspace analysis. To emulate your compiler flags, trace your build command or manually specify equivalent Polyspace analysis options. See “Specify Target Environment and Compiler Behavior”.
- Code Prover has a linking policy that is stricter than regular compilers. For instance, if your compiler allows declaration mismatches with specific compiler options, you cannot emulate this linking policy in Code Prover. See “Troubleshoot Compilation and Linking Errors” (Polyspace Code Prover).

Settings

Default: `generic`

`generic`

Analysis allows only standard syntax.

The language standard is determined by your choice for the following options:

- `C standard version (-c-version)`
- `C++ standard version (-cpp-version)`

If you do not specify a standard explicitly, the standard depends on your choice of compiler.

`gnu3.4`

Analysis allows GCC 3.4 syntax.

`gnu4.6`

Analysis allows GCC 4.6 syntax.

`gnu4.7`

Analysis allows GCC 4.7 syntax.

For unsupported GCC extensions, see “Limitations” on page 1-12.

`gnu4.8`

Analysis allows GCC 4.8 syntax.

For unsupported GCC extensions, see “Limitations” on page 1-12.

`gnu4.9`

Analysis allows GCC 4.9 syntax.

For unsupported GCC extensions, see “Limitations” on page 1-12.

`gnu5.x`

Analysis allows GCC 5.1, 5.2, 5.3, and 5.4 syntax.

If you select `gnu5.x`, the option `Target processor type (-target)` shows only a subset of targets that are allowed for a GCC based compiler. For other targets, use the option `Generic target options`.

For unsupported GCC extensions, see “Limitations” on page 1-12.

`gnu6.x`

Analysis allows GCC 6.1, 6.2, and 6.3 syntax.

If you select `gnu6.x`, the option `Target processor type (-target)` shows only a subset of targets that are allowed for a GCC based compiler. For other targets, use the option `Generic target options`.

For unsupported GCC extensions, see “Limitations” on page 1-12.

`gnu7.x`

Analysis allows GCC 7.1, 7.2, and 7.3 syntax.

If you select `gnu7.x`, the option `Target processor type (-target)` shows only a subset of targets that are allowed for a GCC based compiler. For other targets, use the option `Generic target options`.

For unsupported GCC extensions, see “Limitations” on page 1-12.

`clang3.x`

Analysis allows Clang 3.5, 3.6, 3.7, 3.8, and 3.9 syntax.

`clang4.x`

Analysis allows Clang 4.0.0, and 4.0.1 syntax.

`clang5.x`

Analysis allows Clang 5.0.0, and 5.0.1 syntax.

`visual9.0`

Analysis allows Microsoft® Visual C++® 2008 syntax.

`visual10.0`

Analysis allows Microsoft Visual C++ 2010 syntax.

This option implicitly enables the option `-no-stl-stubs`.

`visual11.0`

Analysis allows Microsoft Visual C++ 2012 syntax.

This option implicitly enables the option `-no-stl-stubs`.

`visual12.0`

Analysis allows Microsoft Visual C++ 2013 syntax.

This option implicitly enables the option `-no-stl-stubs`.

`visual14.0`

Analysis allows Microsoft Visual C++ 2015 syntax (supports Microsoft Visual Studio®update 2).

This option implicitly enables the option `-no-stl-stubs`.

`visual15.x`

Analysis allows Microsoft Visual C++ 2017 syntax (supports Microsoft Visual Studio versions 15.0 up to 15.7).

This option implicitly enables the option `-no-stl-stubs`.

`keil`

Analysis allows non-ANSI® C syntax and semantics associated with the Keil products from ARM (www.keil.com).

`iar`

Analysis allows non-ANSI C syntax and semantics associated with the compilers from IAR Systems (www.iar.com).

`armcc`

Analysis allows non-ANSI C syntax and semantics associated with the ARM® v5 compiler.

If you select `armcc`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the ARM v5 compiler. See `ARM v5 Compiler (-compiler armcc)`.

`armclang`

Analysis allows non-ANSI C syntax and semantics associated with the ARM v6 compiler.

If you select `armclang`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the ARM v6 compiler. See `ARM v6 Compiler (-compiler armclang)`.

`codewarrior`

Analysis allows non-ANSI C syntax and semantics associated with the NXP CodeWarrior® compiler.

If you select codewarrior, in the user interface of the Polyspace desktop products, the option Target processor type (-target) shows only the targets that are allowed for the NXP CodeWarrior compiler. See NXP CodeWarrior Compiler (-compiler codewarrior).

cosmic

Analysis allows non-ANSI C syntax and semantics associated with the Cosmic compiler.

If you select cosmic, in the user interface of the Polyspace desktop products, the option Target processor type (-target) shows only the targets that are allowed for the Comic compiler. See Cosmic Compiler (-compiler cosmic).

diab

Analysis allows non-ANSI C syntax and semantics associated with the Wind River® Diab compiler.

If you select diab, in the user interface of the Polyspace desktop products, the option Target processor type (-target) shows only the targets that are allowed for the NXP CodeWarrior compiler. See Diab Compiler (-compiler diab).

greenhills

Analysis allows non-ANSI C syntax and semantics associated with a Green Hills® compiler.

If you select greenhills, in the user interface of the Polyspace desktop products, the option Target processor type (-target) shows only the targets that are allowed for a Green Hills compiler. See Green Hills Compiler (-compiler greenhills).

iar-ew

Analysis allows non-ANSI C syntax and semantics associated with the IAR Embedded Workbench compiler.

If you select iar-ew, in the user interface of the Polyspace desktop products, the option Target processor type (-target) shows only the targets that are allowed for the IAR Embedded Workbench compiler. See IAR Embedded Workbench Compiler (-compiler iar-ew).

microchip

Analysis allows non-ANSI C syntax and semantics associated with the MPLAB XC8 C compiler.

If you select microchip, in the user interface of the Polyspace desktop products, the option Target processor type (-target) shows only the targets that are allowed for the MPLAB XC8 C compiler. See MPLAB XC8 C Compiler (-compiler microchip).

renesas

Analysis allows non-ANSI C syntax and semantics associated with the Renesas® compiler.

If you select renesas, in the user interface of the Polyspace desktop products, the option Target processor type (-target) shows only the targets that are allowed for the Renesas compiler. See Renesas Compiler (-compiler renesas).

tasking

Analysis allows non-ANSI C syntax and semantics associated with the TASKING compiler.

If you select tasking,in the user interface of the Polyspace desktop products, the option Target processor type (-target) shows only the targets that are allowed for the TASKING compiler. See TASKING Compiler (-compiler tasking).

ti

Analysis allows non-ANSI C syntax and semantics associated with the Texas Instruments™ compiler.

If you select `ti`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the Texas Instruments compiler. See `Texas Instruments Compiler (-compiler ti)`.

`cosmic`

Analysis allows non-ANSI C syntax and semantics associated with the compiler used in the Cosmic software development tools.

If you select `cosmic`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the Cosmic compiler.

Tips

- Your compiler specification determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.
  - To override the macro definition, use the option `Preprocessor definitions (-D)`.
  - To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.
- If you use a Visual Studio compiler, you must use a `Target processor type (-target)` option that sets `long long` to 64 bits. Compatible targets include: `i386`, `sparc`, `m68k`, `powerpc`, `tms320c3x`, `sharc21x61`, `mpc5xx`, `x86_64`, or `mcpu` with `long long` set to 64 (`-long-long-is-64bits` at the command line).
- If you use the option `Check JSF AV C++ rules (-jsf-coding-rules)`, select the compiler `generic`. If you use another compiler, Polyspace cannot check the JSF® coding rules that require conforming to the ISO standard. For example, AV Rule 8: “All code shall conform to ISO/IEC 14882:2002(E) standard C++.”

Limitations

Polyspace does not support certain features of these compilers:

- GNU® compilers (version 4.7 or later):
  - Nested functions.

    For instance, the function `bar` is nested in function `foo`:

    ```
    int foo (int a, int b)
    {
      int bar (int c) { return c * c; }

      return bar (a) + bar (b);
    }
    ```
  - Binary operations with vector types where one operand uses the shorthand notation for uniform vectors.

    For instance, in the addition operation, `2+a`, 2 is used as a shorthand notation for {2,2,2,2}.

    ```
    typedef int v4si __attribute__ ((vector_size (16)));
    v4si res, a = {1,2,3,4};

    res = 2 + a;  /* means {2,2,2,2} + a  */
    ```

- Forward declaration of function parameters.

  For instance, the parameter `len` is forward declared:

  ```
  void func (int len; char data[len][len], int len)
  {
    /* … */
  }
  ```

- Complex integer data types.

  However, complex floating point data types are supported.

- Initialization of structures with flexible array members using an initialization list.

  For instance, the structure `S` has a flexible array member `tab`. A variable of type `S` is directly initialized with an initialization list.

  ```
  struct S {
      int x;
      int tab[];            /* flexible array member - not supported */
  };
  struct S s = { 0, 1, 2} ;
  ```

  You see a warning during analysis and a red check in the results when you dereference, for instance, `s.tab[1]`.

- 128-bit variables.

  Polyspace cannot analyze this data type semantically. Bug Finder allows use of 128-bit data types, but Code Prover shows a compilation error if you use such a data type, for instance, the GCC extension `__float128`.

- GNU compilers version 7.x:

  - Type names `_FloatN` and `_FloatNx` are not semantically supported. The analysis treats them as type `float`, `double`, or `long double`.
  - Constants of type `_FloatN` or `_FloatNx` with suffixes `fN`, `FN`, or `fNx`, such as `1.2f123` or `2.3F64x` are not supported.

- Visual Studio compilers:

  - C++ Accelerated Massive Parallelism (AMP).

    C++ AMP is a Visual Studio feature that accelerates your C++ code execution for certain types of data-parallel hardware on specific targets. You typically use the `restrict` keyword to enable this feature.

    ```
    void Buffer() restrict(amp)
    {
      ...
    }
    ```

  - `__assume` statements.

    You typically use `__assume` with a condition that is false. The statement indicates that the optimizer must assume the condition to be henceforth true. Code Prover cannot reconcile this contradiction. You get the error:

    ```
    Asked for compulsory presence of absent entity : assert
    ```

- Managed Extensions for C++ (required for the .NET Framework), or its successor, C++/CLI (C ++ modified for Common Language Infrastructure)
- `__declspec` keyword with attributes other than `noreturn`, `nothrow`, `selectany` or `thread`.

Command-Line Information

Parameter: `-compiler`
Value: `generic | gnu3.4 | gnu4.6 | gnu4.7 | gnu4.8 | gnu4.9 | gnu5.x | gnu6.x | gnu7.x | clang3.x | clang4.x | clang5.x | visual9.0 | visual10.0 | visual11.0 | visual12.0 | visual14.0 | visual15.x | keil | iar | armcc | armclang | codewarrior | cosmic | diab | greenhills | iar-ew | microchip |renesas | tasking | ti`
Default: `generic`
Example 1 (Bug Finder): `polyspace-bug-finder -lang c -sources "file1.c,file2.c" -compiler gnu4.6`
Example 2 (Bug Finder): `polyspace-bug-finder -lang cpp -sources "file1.cpp,file2.cpp" -compiler visual9.0`
Example 1 (Code Prover): `polyspace-code-prover -lang c -sources "file1.c,file2.c" -lang c -compiler gnu4.6`
Example 2 (Code Prover): `polyspace-code-prover -lang cpp -sources "file1.cpp,file2.cpp" -compiler visual9.0`
Example 1 (Bug Finder Server): `polyspace-bug-finder-server -lang c -sources "file1.c,file2.c" -compiler gnu4.6`
Example 2 (Bug Finder Server): `polyspace-bug-finder-server -lang cpp -sources "file1.cpp,file2.cpp" -compiler visual9.0`
Example 1 (Code Prover Server): `polyspace-code-prover-server -lang c -sources "file1.c,file2.c" -lang c -compiler gnu4.6`
Example 2 (Code Prover Server): `polyspace-code-prover-server -lang cpp -sources "file1.cpp,file2.cpp" -compiler visual9.0`

See Also

`C standard version (-c-version)` | `C++ standard version (-cpp-version)` | `Target processor type (-target)`

Topics

“Specify Polyspace Analysis Options”
“Troubleshoot Compilation Errors”
“Specify Target Environment and Compiler Behavior”
“Supported Keil or IAR Language Extensions”

Target processor type (-target)

Specify size of data types and endianness by selecting a predefined target processor

Description

Specify the processor on which you deploy your code.

The target processor determines the sizes of fundamental data types and the endianness of the target machine. You can analyze code intended for an unlisted processor type by using one of the other processor types, if they share common data properties.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. To see the sizes of types, click the **Edit** button to the right of the **Target processor type** drop-down list.

For some compilers, in the user interface, you see only the processors allowed for that compiler. For these compilers, you also cannot see the data type sizes in the user interface. See the links in the table below for the data type sizes.

Command line: Use the option `-target`. See “Command-Line Information” on page 1-17.

Why Use This Option

You specify a target processor so that some of the Polyspace run-time checks are tailored to the data type sizes and other properties of that processor.

For instance, a variable can overflow for smaller values on a 32-bit processor such as i386 compared to a 64-bit processor such as x86_64. If you select x86_64 for your Polyspace analysis, but deploy your code to the i386 processor, your Polyspace results are not always applicable.

Once you select a target processor, you can specify if the default sign of char is signed or unsigned. To determine which signedness to specify, compile this code using the compiler settings that you typically use:

```
#include <limits.h>
int array[(char)UCHAR_MAX]; /* If char is signed, the array size is -1
```

If the code compiles, the default sign of char is unsigned. For instance, on a GCC compiler, the code compiles with the `-fsigned-char` flag and fails to compile with the `-funsigned-char` flag.

Settings

Default: `i386`

This table shows the size of each fundamental data type that Polyspace considers. For some targets, you can modify the default size by clicking the **Edit** button to the right of the **Target processor type** drop-down list. The optional values for those targets are shown in [brackets] in the table.

Target	char	short	int	long	long long	float	double	long double[a]	ptr	Default sign of char	endian	Alignment
i386	8	16	32	32	64	32	64	96	32	signed	Little	32
sparc	8	16	32	32	64	32	64	128	32	signed	Big	64
m68k[b]	8	16	32	32	64	32	64	96	32	signed	Big	64
powerpc	8	16	32	32	64	32	64	128	32	unsigned	Big	64
c-167	8	16	16	32	32	32	64	64	16	signed	Little	64
tms320c3x	32	32	32	32	64	32	32	64	32	signed	Little	32
sharc21x61	32	32	32	32	64	32	32 [64]	32 [64]	32	signed	Little	32
necv850	8	16	32	32	32	32	32	64	32	signed	Little	32 [16, 8]
hc08[c]	8	16	16 [32]	32	32	32	32 [64]	32 [64]	16[d]	unsigned	Big	32 [16]
hc12	8	16	16 [32]	32	32	32	32 [64]	32 [64]	32[6]	signed	Big	32 [16]
mpc5xx	8	16	32	32	64	32	32 [64]	32 [64]	32	signed	Big	32 [16]
c18	8	16	16	32 [24] [e]	32	32	32	32	16 [24]	signed	Little	8
x86_64	8	16	32	64 [32][f]	64	32	64	128	64	signed	Little	64 [32]
mcpu... (Advanced)[g]	8 [16]	8 [16]	16 [32]	32	32 [64]	32	32 [64]	32 [64]	16 [32]	signed	Little	32 [16, 8]
Targets for ARM v5 compiler	See `ARM v5 Compiler (-compiler armcc)`.											
Targets for ARM v6 compiler	See `ARM v6 Compiler (-compiler armclang)`.											
Targets for NPX CodeWarrior compiler	See `NXP CodeWarrior Compiler (-compiler codewarrior)`.											
Targets for Cosmic compiler	See `Cosmic Compiler (-compiler cosmic)`.											
Targets for Diab compiler	See `Diab Compiler (-compiler diab)`.											
Targets for Green Hills compiler	See `Green Hills Compiler (-compiler greenhills)`.											

Target	char	short	int	long	long long	float	double	long double[a]	ptr	Default sign of char	endian	Alignment
Targets for IAR Embedded Workbench compiler	See IAR Embedded Workbench Compiler (-compiler iar-ew).											
Targets for MPLAB XC8 C compiler	See MPLAB XC8 C Compiler (-compiler microchip)											
Targets for Renesas compiler	See Renesas Compiler (-compiler renesas).											
Targets for TASKING compiler	See TASKING Compiler (-compiler tasking).											
Targets for Texas Instruments compiler	See Texas Instruments Compiler (-compiler ti).											

a. For targets where the size of `long double` is greater than 64 bits, the size used for computations is not always the same as the size listed in this table. The exceptions are:

- For targets `i386`, `x86_64` and `m68k`, 80 bits are used for computations, following the practice in common compilers.
- For the target `tms320c3x`, 40 bits are used for computation, following the TMS320C3x specifications.
- If you use a Visual compiler, the size of `long double` used for computations is the same as size of `double`, following the specification of Visual C++ compilers.

b. The M68k family (68000, 68020, and so on) includes the “ColdFire” processor

c. Non-ANSI C specified keywords and compiler implementation-dependent pragmas and interrupt facilities are not taken into account by this support

d. All kinds of pointers (near or far pointer) have 2 bytes (hc08) or 4 bytes (hc12) of width physically.

e. The `c18` target supports the type `short long` as 24 bits in size.

f. Use option `-long-is-32bits` to support Microsoft C/C++ Win64 target.

g. `mcpu` is a reconfigurable Micro Controller/Processor Unit target. You can use this type to configure one or more generic targets. For more information, see `Generic target options`.

Tips

If your processor is not listed, use a similar processor that shares the same characteristics, or create an `mcpu` generic target processor. See `Generic target options`.

You can also create a custom target by explicitly stating sizes of fundamental types and so on with the option `-custom-target`.

Command-Line Information

Parameter: `-target`
Value: `i386 | sparc | m68k | powerpc | c-167 | tms320c3x | sharc21x61 | necv850 | hc08 | hc12 | mpc5xx | c18 | x86_64 | mcpu`
Default: `i386`
Example (Bug Finder): `polyspace-bug-finder -target m68k`
Example (Code Prover): `polyspace-code-prover -target m68k`

Example (Bug Finder Server): `polyspace-bug-finder-server -target m68k`
Example (Code Prover Server): `polyspace-code-prover-server -target m68k`

You can override the default values for some targets by using specific command-line options. See the section **Command-Line Options** in `Generic target options`.

See Also

Polyspace Analysis Options
`-custom-target`

Polyspace Results
`Higher Estimate of Local Variable Size|Lower Estimate of Local Variable Size`

Topics
“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”

ARM v5 Compiler (-compiler armcc)

Specify ARM v5 compiler

Description

Specify armcc for the Compiler (-compiler) option if you compile your code with a ARM v5 compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select armcc for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a ARM v5 compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine, and certain keyword definitions.

If you specify the armcc compiler, you must specify the path to your compiler header files. See "Provide Standard Library Headers for Polyspace Analysis".

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option -dump-preprocessing-info.

- To override the macro definition, use the option Preprocessor definitions (-D).
- To undefine a macro, use the option Disabled preprocessor definitions (-U).

Command-Line Information

Parameter: -compiler armcc -target
Value: arm
Default: arm
Example (Bug Finder): polyspace-bug-finder -compiler armcc -target arm
Example (Code Prover): polyspace-code-prover -compiler armcc -target arm
Example (Bug Finder Server): polyspace-bug-finder-server -compiler armcc -target arm
Example (Code Prover Server): polyspace-code-prover-server -compiler armcc -target arm

See Also

Compiler (-compiler) | Target processor type (-target)

Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

Introduced in R2019a

ARM v6 Compiler (`-compiler armclang`)

Specify ARM v6 compiler

Description

Specify `armclang` for the `Compiler (-compiler)` option if you compile your code with a ARM v6 compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `armclang` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a ARM v6 compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine, and certain keyword definitions.

If you specify the `armclang` compiler, you must specify the path to your compiler header files. See "Provide Standard Library Headers for Polyspace Analysis".

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Command-Line Information

Parameter: `-compiler armclang -target`
Value: `arm | arm64`
Default: `arm`
Example (Bug Finder): `polyspace-bug-finder -compiler armclang -target arm64`
Example (Code Prover): `polyspace-code-prover -compiler armclang -target arm64`
Example (Bug Finder Server): `polyspace-bug-finder-server -compiler armclang -target arm64`
Example (Code Prover Server): `polyspace-code-prover-server -compiler armclang -target arm64`

See Also

`Compiler (-compiler)` | `Target processor type (-target)`

Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

Introduced in R2019a

NXP CodeWarrior Compiler (-compiler codewarrior)

Specify NXP CodeWarrior compiler

Description

Specify codewarrior for Compiler (-compiler) if you compile your code using a NXP CodeWarrior compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select codewarrior for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a NXP CodeWarrior compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the codewarrior compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option -dump-preprocessing-info.

- To override the macro definition, use the option Preprocessor definitions (-D).
- To undefine a macro, use the option Disabled preprocessor definitions (-U).

Command-Line Information

Parameter: -compiler codewarrior -target
Value: s12z | powerpc
Default: s12z
Example (Bug Finder): polyspace-bug-finder -compiler codewarrior -target powerpc
Example (Code Prover): polyspace-code-prover -compiler codewarrior -target powerpc
Example (Bug Finder Server): polyspace-bug-finder-server -compiler codewarrior -target powerpc
Example (Code Prover Server): polyspace-code-prover-server -compiler codewarrior -target powerpc

See Also

Compiler (-compiler) | Target processor type (-target)

Topics
“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”

Introduced in R2018a

Cosmic Compiler (-compiler cosmic)

Specify Cosmic compiler

Description

Specify `cosmic` for the `Compiler (-compiler)` option if you compile your code with a Cosmic compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `cosmic` for **Compiler**, in the user interface, you see only the processors allowed for a Cosmic compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine, and certain keyword definitions.

If you specify the `cosmic` compiler, you must specify the path to your compiler header files. See "Provide Standard Library Headers for Polyspace Analysis".

Settings

To see the default sizes in bits for the fundamental types that the target uses, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Command-Line Information

Parameter: `-compiler cosmic -target`
Value: `s12z`
Default: `s12z`
Example (Bug Finder): `polyspace-bug-finder -compiler cosmic -target s12z`
Example (Code Prover): `polyspace-code-prover -compiler cosmic -target s12z`
Example (Bug Finder Server): `polyspace-bug-finder-server -compiler cosmic -target s12z`
Example (Code Prover Server): `polyspace-code-prover-server -compiler cosmic -target s12z`

See Also

`Compiler (-compiler)` | `Target processor type (-target)`

Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

Introduced in R2019b

Diab Compiler (`-compiler diab`)

Specify the Wind River Diab compiler

Description

Specify `diab` for `Compiler (-compiler)` if you compile your code using the Wind River Diab compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `diab` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for the Diab compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `diab` compiler, you must specify the path to your compiler header files. See "Provide Standard Library Headers for Polyspace Analysis".

The software supports version 5.9.6 and older versions of the Diab compiler.

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Tips

If you encounter errors during Polyspace analysis, see "Errors Related to Diab Compiler".

Command-Line Information

Parameter: `-compiler diab -target`
Value: `i386 | powerpc | arm | coldfire | mips | mcore | rh850 | superh | tricore`
Default: `powerpc`
Example (Bug Finder): `polyspace-bug-finder -compiler diab -target tricore`
Example (Code Prover): `polyspace-code-prover -compiler diab -target tricore`
Example (Bug Finder Server): `polyspace-bug-finder-server -compiler diab -target tricore`
Example (Code Prover Server): `polyspace-code-prover-server -compiler diab -target tricore`

See Also

`Compiler (-compiler)` | `Target processor type (-target)`

Topics
“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”

Introduced in R2016b

Green Hills Compiler (`-compiler greenhills`)

Specify Green Hills compiler

Description

Specify `greenhills` for `Compiler (-compiler)` if you compile your code using a Green Hills compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `greenhills` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a Green Hills compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `greenhills` compiler, you must specify the path to your compiler header files. See "Provide Standard Library Headers for Polyspace Analysis".

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Tips

- If you encounter errors during a Polyspace analysis, see "Errors Related to Green Hills Compiler"
- Polyspace supports the embedded configuration for the i386 target. If your x86 Green Hills compiler is configured for native Windows® development, you can see compilation errors or incorrect analysis results with Code Prover. Contact Technical Support.

  For instance, Green Hills compilers consider a size of 12 bytes for `long double` for embedded targets, but 8 bytes for native Windows. Polyspace considers 12 bytes by default.
- If you create a Polyspace project from a build command that uses a Green Hills compiler, the compiler options `-filetype` and `-os_dir` are not implemented in the project. To emulate the `-os_dir` option, you can explicitly add the path argument of the option as an include folder to your Polyspace project.

Command-Line Information

Parameter: `-compiler greenhills -target`
Value: `powerpc | powerpc64 | arm | arm64 | tricore | rh850 | arm | i386 | x86_64`
Default: `powerpc`

Example (Bug Finder): `polyspace-bug-finder -compiler greenhills -target arm`
Example (Code Prover): `polyspace-code-prover -compiler greenhills -target arm`
Example (Bug Finder Server): `polyspace-bug-finder-server -compiler greenhills -target arm`
Example (Code Prover Server): `polyspace-code-prover-server -compiler greenhills -target arm`

See Also

`Compiler (-compiler)` | `Target processor type (-target)`

Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

Introduced in R2017b

IAR Embedded Workbench Compiler (`-compiler iar-ew`)

Specify IAR Embedded Workbench compiler

Description

Specify `iar-ew` for `Compiler (-compiler)` if you compile your code using a IAR Embedded Workbench compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `iar-ew` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a IAR Embedded Workbench compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `iar-ew` compiler, you must specify the path to your compiler header files. See "Provide Standard Library Headers for Polyspace Analysis".

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Tips

Polyspace does not support some constructs specific to the IAR compiler.

For the list of unsupported constructs, see `codeprover_limitations.pdf` in *`polyspaceroot`* `\polyspace\verifier\code_prover_desktop`. Here, *`polyspaceroot`* is the MATLAB® installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

Command-Line Information

Parameter: `-compiler iar-ew -target`
Value: `arm | avr | msp430 | rh850 | rl78`
Default: `arm`
Example (Bug Finder): `polyspace-bug-finder -compiler iar-ew -target rl78`
Example (Code Prover): `polyspace-code-prover -compiler iar-ew -target rl78`
Example (Bug Finder Server): `polyspace-bug-finder-server -compiler iar-ew -target rl78`
Example (Code Prover Server): `polyspace-code-prover-server -compiler iar-ew -target rl78`

See Also

`Compiler (-compiler)` | `Target processor type (-target)`

Topics

“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”

Introduced in R2018a

MPLAB XC8 C Compiler (`-compiler microchip`)

Specify MPLAB XC8 C compiler

Description

Specify `microchip` for the `Compiler (-compiler)` option if you compile your code with a MPLAB XC8 C compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `microchip` for **Compiler**, in the user interface, you see only the processors allowed for a MPLAB XC8 C compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine, and certain keyword definitions.

If you specify the `microchip` compiler, you must specify the path to your compiler header files. See "Provide Standard Library Headers for Polyspace Analysis".

Settings

To see the default sizes in bits for the fundamental types that the target uses, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Tip

Polyspace does not support the Atmel families of processors, such as AVR, TinyAVR, MegaAVR, XMEGA, and SAM32.

Command-Line Information

Parameter: `-compiler microchip -target`
Value: `pic`
Default: `pic`
Example (Bug Finder): `polyspace-bug-finder -compiler microchip -target pic`
Example (Code Prover): `polyspace-code-prover -compiler microchip -target pic`
Example (Bug Finder Server): `polyspace-bug-finder-server -compiler microchip -target pic`
Example (Code Prover Server): `polyspace-code-prover-server -compiler microchip -target pic`

See Also

`Compiler (-compiler)` | `Target processor type (-target)`

Topics
“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”

Introduced in R2020a

Renesas Compiler (`-compiler renesas`)

Specify Renesas compiler

Description

Specify `renesas` for the `Compiler (-compiler)` option if you compile your code with a Renesas compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `renesas` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a Renesas compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine, and certain keyword definitions.

If you specify the `renesas` compiler, you must specify the path to your compiler header files. See "Provide Standard Library Headers for Polyspace Analysis".

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Command-Line Information

Parameter: `-compiler renesas -target`
Value: `rl78 | rh850 | rx`
Default: `rl78`
Example (Bug Finder): `polyspace-bug-finder -compiler renesas -target rx`
Example (Code Prover): `polyspace-code-prover -compiler renesas -target rx`
Example (Bug Finder Server): `polyspace-bug-finder-server -compiler renesas -target rx`
Example (Code Prover Server): `polyspace-code-prover-server -compiler renesas -target rx`

See Also

`Compiler (-compiler)` | `Target processor type (-target)`

Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

Introduced in R2018b

TASKING Compiler (-compiler tasking)

Specify the Altium TASKING compiler

Description

Specify `tasking` for `Compiler (-compiler)` if you compile your code using the Altium® TASKING compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `tasking` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for the TASKING compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `tasking` compiler, you must specify the path to your compiler header files. See "Provide Standard Library Headers for Polyspace Analysis".

The software supports different versions of the TASKING compiler, depending on the target:

- TriCore: 6.0 and older versions
- C166: 4.0 and older versions
- ARM: 5.2 and older versions
- RH850: 2.2 and older versions

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Tips

- Polyspace does not support some constructs specific to the TASKING compiler.

  For the list of unsupported constructs, see `codeprover_limitations.pdf` in *`polyspaceroot`*`\polyspace\verifier\code_prover_desktop`. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.
- The CPU used is TC1793. If you use a different CPU, set the following analysis options in your project:
  - `Disabled preprocessor definitions (-U)`: Undefine the macro `__CPU_TC1793B__`.
  - `Preprocessor definitions (-D)`: Define the macro `__CPU__`. Enter `__CPU__=`*`xxx`*, where *`xxx`* is the name of your CPU.

Additionally, define the equivalent of the macro `__CPU_TC1793B__` for your CPU. For instance, enter `__CPU_TC1793A__`.

Instead of manually specifying your compiler, if you trace your build command (makefile), Polyspace can detect your CPU and add the required definitions in your project.

- For some errors related to TASKING compiler-specific constructs, see solutions in “Errors Related to TASKING Compiler”.

Command-Line Information

Parameter: `-compiler tasking -target`
Value: `tricore | c166 | rh850 | arm`
Default: `tricore`
Example (Bug Finder): `polyspace-bug-finder -compiler tasking -target tricore`
Example (Code Prover): `polyspace-code-prover -compiler tasking -target tricore`
Example (Bug Finder Server): `polyspace-bug-finder-server -compiler tasking -target tricore`
Example (Code Prover Server): `polyspace-code-prover-server -compiler tasking -target tricore`

See Also

`Compiler (-compiler)` | `Target processor type (-target)`

Topics

“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”

Introduced in R2017a

Texas Instruments Compiler (-compiler ti)

Specify Texas Instruments compiler

Description

Specify `ti` for `Compiler (-compiler)` if you compile your code using a Texas Instruments compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `ti` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a Texas Instruments compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `ti` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Tips

Polyspace does not support some constructs specific to the Texas Instruments compiler.

For the list of unsupported constructs, see `codeprover_limitations.pdf` in *`polyspaceroot`*`\polyspace\verifier\code_prover_desktop`. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

Command-Line Information

Parameter: `-compiler ti -target`
Value: `c28x | c6000 | arm | msp430`
Default: `c28x`
Example (Bug Finder): `polyspace-bug-finder -compiler ti -target msp430`
Example (Code Prover): `polyspace-code-prover -compiler ti -target msp430`
Example (Bug Finder Server): `polyspace-bug-finder-server -compiler ti -target msp430`
Example (Code Prover Server): `polyspace-code-prover-server -compiler ti -target msp430`

See Also

`Compiler (-compiler)` | `Target processor type (-target)`

Topics

“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”

Introduced in R2018a

Generic target options

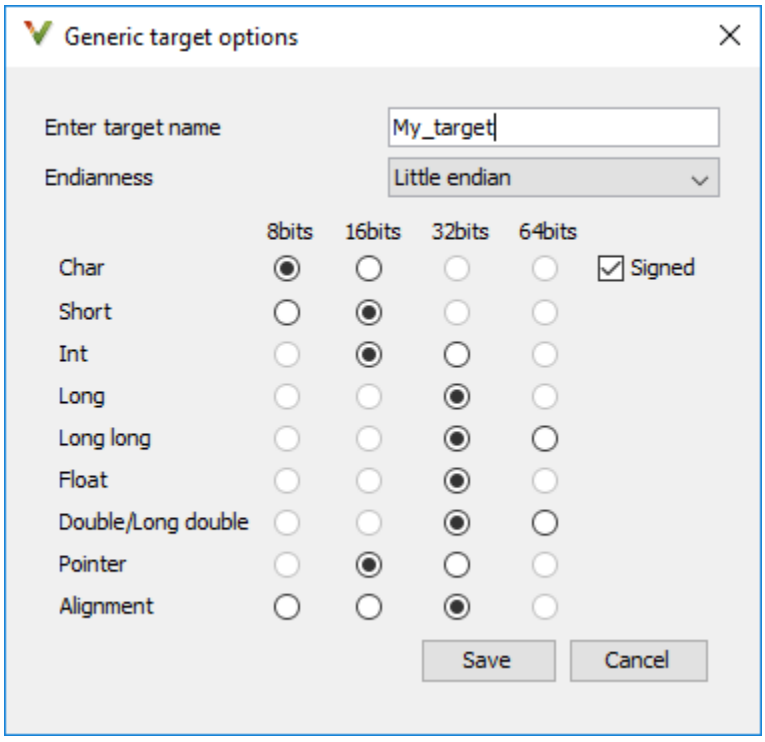
Specify size of data types and endianness by creating your own target processor

Description

If a target processor is not directly supported by Polyspace, you can create your own target. You specify the target `mcpu` representing a generic "Micro Controller/Processor Unit" and then explicitly specify sizes of fundamental data types, endianness and other characteristics.

Settings

In the user interface of the Polyspace desktop products, the **Generic target options** dialog box opens when you set the **Target processor type** to `mcpu`. The **Target processor type** option is available on the **Target & Compiler** node in the **Configuration** pane.



Use the dialog box to specify the name of a new `mcpu` target, for example `My_target`. That new target is added to the **Target processor type** option list.

Default characteristics of a new target: listed as *type* `[size]`

- *char* `[8]`
- *short* `[16]`
- *int* `[16]`
- *long* `[32]`

- *long long* [32]
- *float* [32]
- *double* [32]
- *long double* [32]
- *pointer* [16]
- *alignment* [32]
- *char* is signed
- *endianness* is little-endian

Dependency

A custom target can only be created when `Target processor type (-target)` is set to `mcpu`.

A custom target is not available when `Compiler (-compiler)` is set to one of the `visual*` options.

Command-Line Options

When using the command line, use `-target mcpu` along with these target specification options.

Option	Description	Available With	Example
`-little-endian`	Little-endian architectures are Less Significant byte First (LSF). For example: i386. Specifies that the less significant byte of a short integer (e.g. 0x00FF) is stored at the first byte (0xFF) and the most significant byte (0x00) at the second byte.	`mcpu`	`polyspace-bug-finder -target mcpu -little-endian`

Option	Description	Available With	Example
`-big-endian`	Big-endian architectures are Most Significant byte First (MSF). For example: SPARC, m68k. Specifies that the most significant byte of a short integer (e.g. 0x00FF) is stored at the first byte (0x00) and the less significant byte (0xFF) at the second byte.	`mcpu`	`polyspace-bug-finder -target mcpu -big-endian`
`-default-sign-of-char [signed | unsigned]`	Specify default sign of `char`. `signed`: Specifies that `char` is signed, overriding target's default. `unsigned`: Specifies that `char` is unsigned, overriding target's default.	All targets	`polyspace-bug-finder -default-sign-of-char unsigned -target mcpu`
`-char-is-16bits`	`char` defined as 16 bits and all objects have a minimum alignment of 16 bits Incompatible with `-short-is-8bits` and `-align 8`	`mcpu`	`polyspace-bug-finder -target mcpu -char-is-16bits`
`-short-is-8bits`	Define `short` as 8 bits, regardless of sign	`mcpu`	`polyspace-bug-finder -target mcpu -short-is-8bits`
`-int-is-32bits`	Define `int` as 32 bits, regardless of sign. Alignment is also set to 32 bits.	`mcpu`, `hc08`, `hc12`, `mpc5xx`	`polyspace-bug-finder -target mcpu -int-is-32bits`
`-long-is-32bits`	Define `long` as 32 bits, regardless of sign. Alignment is also set to 32 bits. If your project sets `int` to 64 bits, you cannot use this option.	All targets	`polyspace-bug-finder -target mcpu -long-is-32bits`

Option	Description	Available With	Example
`-long-long-is-64bits`	Define `long long` as 64 bits, regardless of sign. Alignment is also set to 64 bits.	`mcpu`	`polyspace-bug-finder -target mcpu -long-long-is-64bits`
`-double-is-64bits`	Define `double` and `long double` as 64 bits, regardless of sign.	`mcpu`, `sharc21x61`, `hc08`, `hc12`, `mpc5xx`	`polyspace-bug-finder -target mcpu -double-is-64bits`
`-pointer-is-24bits`	Define pointer as 24 bits, regardless of sign.	`c18`	`polyspace-bug-finder -target c18 -pointer-is-24bits`
`-pointer-is-32bits`	Define pointer as 32 bits, regardless of sign.	`mcpu`	`polyspace-bug-finder -target mcpu -pointer-is-32bits`
`-align [32|16|8]`	Specifies the largest alignment of struct or array objects to the 32, 16 or 8 bit boundaries. Consequently, the array or struct storage is strictly determined by the size of the individual data objects without member and end padding.	`mcpu`, `hc08`, `hc12`, `mpc5xx`. Other than `mcpu`, all targets support only 16 or 32 bits.	`polyspace-bug-finder -target mcpu -align 16`

See also:

- `Management of wchar_t (-wchar-t-type-is)`
- `Management of size_t (-size-t-type-is)`
- `Enum type definition (-enum-type-definition)`

You can also use the option `-custom-target` to specify sizes in bytes of fundamental data types, signedness of plain `char`, alignment of structures and underlying types of standard `typedef`-s such as `size_t`, `wchar_t` and `ptrdiff_t`.

Examples

Targets for GCC Based Compilers

If you select one of the `gnu#.x` compilers for `Compiler (-compiler)`, you can specify one of the supported target processor types. See `Target processor type (-target)`. If a target processor type is not directly listed as supported, you can create the target by using this option.

For instance, you can create these targets:

- **Tricore**: Use these options:

  ```
  -target mcpu
  -int-is-32bits
  ```

```
-long-long-is-64bits
-double-is-64bits
-pointer-is-32bits
-enum-type-definition auto-signed-first
-wchar-t-type-is signed-int
```

- **PowerPC**: Use these options:

  ```
  -target mcpu
  -int-is-32bits
  -long-long-is-64bits
  -double-is-64bits
  -pointer-is-32bits
  -wchar-t-type-is signed-int
  ```

- **ARM**: Use these options:

  ```
  -target mcpu
  -int-is-32bits
  -long-long-is-64bits
  -double-is-64bits
  -pointer-is-32bits
  -enum-type-definition auto-signed-first
  -wchar-t-type-is unsigned-int
  ```

- **MSP430**: Use these options:

  ```
  -target mcpu
  -long-long-is-64bits
  -double-is-64bits
  -wchar-t-type-is signed-long
  -align 16
  ```

See Also

`Target processor type (-target)`

Topics

“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”

Sfr type support (-sfr-types)

Specify sizes of `sfr` types for code developed with Keil or IAR compilers

Description

Specify sizes of `sfr` types (types that define special function registers).

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See “Dependency” on page 1-42 for other options you must also enable.

Command line: Use the option `-sfr-types`. See “Command-Line Information” on page 1-42.

Why Use This Option

Use this option if you have statements such as `sfr addr = 0x80;` in your code. `sfr` types are not standard C types. Therefore, you must specify their sizes explicitly for the Polyspace analysis.

Settings

No Default

List each sfr name and its size in bits.

Dependency

This option is available only when `Compiler (-compiler)` is set to `keil` or `iar`.

Command-Line Information

Syntax: `-sfr-types` *`sfr_name=size_in_bits`*`,...`
No Default
Name Value: an sfr name such as `sfr16`.
Size Value: `8 | 16 | 32`
Example (Bug Finder): `polyspace-bug-finder -lang c -compiler iar -sfr-types sfr=8,sfr16=16 ...`
Example (Code Prover): `polyspace-code-prover -lang c -compiler iar -sfr-types sfr=8,sfr16=16 ...`
Example (Bug Finder Server): `polyspace-bug-finder-server -lang c -compiler iar -sfr-types sfr=8,sfr16=16 ...`
Example (Code Prover Server): `polyspace-code-prover-server -lang c -compiler iar -sfr-types sfr=8,sfr16=16 ...`

See Also

Topics

“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”

“Supported Keil or IAR Language Extensions”

Division round down (-div-round-down)

Round down quotients from division or modulus of negative numbers instead of rounding up

Description

Specify whether quotients from division and modulus of negative numbers are rounded up or down.

Note `a = (a / b) * b + a % b` is always true.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-div-round-down`. See “Command-Line Information” on page 1-45.

Why Use This Option

Use this option to emulate your compiler.

The option is relevant only for compilers following C90 standard (ISO/IEC 9899:1990). The standard stipulates that "*if either operand of / or % is negative, whether the result of the / operator, is the largest integer less or equal than the algebraic quotient or the smallest integer greater or equal than the quotient, is implementation defined, same for the sign of the % operator*". The standard allows compilers to choose their own implementation.

For compilers following the C99 standard ((ISO/IEC 9899:1999), this option is not required. The standard enforces division with rounding towards zero (section 6.5.5).

Settings

☑ On

If either operand `/` or `%` is negative, the result of the `/` operator is the largest integer less than or equal to the algebraic quotient. The result of the `%` operator is deduced from `a % b = a - (a / b) * b`.

Example: `assert(-5/3 == -2 && -5%3 == 1);` is true.

☐ Off (default)

If either operand of `/` or `%` is negative, the result of the `/` operator is the smallest integer greater than or equal to the algebraic quotient. The result of the `%` operator is deduced from `a % b = a - (a / b) * b`.

This behavior is also known as rounding towards zero.

Example: `assert(-5/3 == -1 && -5%3 == -2);` is true.

Command-Line Information

Parameter: `-div-round-down`
Default: Off
Example (Bug Finder): `polyspace-bug-finder -div-round-down`
Example (Code Prover): `polyspace-code-prover -div-round-down`
Example (Bug Finder Server): `polyspace-bug-finder-server -div-round-down`
Example (Code Prover Server): `polyspace-code-prover-server -div-round-down`

See Also

Topics

“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”

Enum type definition (-enum-type-definition)

Specify how to represent an `enum` with a base type

Description

Allow the analysis to use different base types to represent an enumerated type, depending on the enumerator values and the selected definition. When using this option, each `enum` type is represented by the smallest integral type that can hold its enumeration values.

This option is available on the **Target & Compiler** node in the **Configuration** pane.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-enum-type-definition`. See “Command-Line Information” on page 1-47.

Why Use This Option

Your compiler represents `enum` variables as constants of a base integer type. Use this option so that you can emulate your compiler.

To check your compiler settings:

1. Compile this code using the compiler settings that you typically use:

   ```
   enum { MAXSIGNEDBYTE=127 } mysmallenum_t;

   int dummy[(int)sizeof(mysmallenum_t) - (int)sizeof(int)];
   ```

   If compilation fails, you have to use one of `auto-signed-first` or `auto-unsigned-first`.
2. Compile this code using the compiler settings that you typically use:

   ```
   #include <limits.h>

   enum { MYINTMAX = INT_MAX } myintenum_t;

   int dummy[(MYINTMAX + 1) < 0 ? -1:1];
   ```

   If compilation fails, use `auto-signed-first` for this option, otherwise use `auto-unsigned-first`.

Settings

Default: `defined-by-compiler`

`defined-by-compiler`

Uses the signed integer type for all compilers except gnu, clang and tasking.

For the gnu and clang compilers, it uses the first type that can hold all of the enumerator values from this list: `unsigned int`, `signed int`, `unsigned long`, `signed long`, `unsigned long long` and `signed long long`.

For the tasking compiler, it uses the first type that can hold all of the enumerator values from this list: `char`, `unsigned char`, `short`, `unsigned short`, `int`, and `unsigned int`.

`auto-signed-first`

Uses the first type that can hold all of the enumerator values from this list: `signed char`, `unsigned char`, `signed short`, `unsigned short`, `signed int`, `unsigned int`, `signed long`, `unsigned long`, `signed long long`, and `unsigned long long`.

`auto-unsigned-first`

Uses the first type that can hold all of the enumerator values from these lists:

- If enumerator values are positive: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`.
- If one or more enumerator values are negative: `signed char`, `signed short`, `signed int`, `signed long`, and `signed long long`.

Command-Line Information

Parameter: `-enum-type-definition`
Value: `defined-by-compiler | auto-signed-first | auto-unsigned-first`
Default: `defined-by-compiler`
Example (Bug Finder): `polyspace-bug-finder -enum-type-definition auto-signed-first`
Example (Code Prover): `polyspace-code-prover -enum-type-definition auto-signed-first`
Example (Bug Finder Server): `polyspace-bug-finder-server -enum-type-definition auto-signed-first`
Example (Code Prover Server): `polyspace-code-prover-server -enum-type-definition auto-signed-first`

See Also

Topics

“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”

Signed right shift (`-logical-signed-right-shift`)

Specify how to treat the sign bit for logical right shifts on signed variables

Description

Choose between arithmetic and logical shift for right shift operations on negative values.

This option does not modify compile-time expressions. For more details, see “Limitation” on page 1-48.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-logical-signed-right-shift`. See “Command-Line Information” on page 1-49.

Why Use This Option

The C99 Standard (sec 6.5.7) states that for a right-shift operation `x1>>x2`, if `x1` is signed and has negative values, the behavior is implementation-defined. Different compilers choose between arithmetic and logical shift. Use this option to emulate your compiler.

Settings

Default: `Arithmetical`

`Arithmetical`

The sign bit remains:

```
(-4) >> 1 = -2
(-7) >> 1 = -4
  7  >> 1 = 3
```

`Logical`

0 replaces the sign bit:

```
(-4) >> 1 = (-4U) >> 1 = 2147483646
(-7) >> 1 = (-7U) >> 1 = 2147483644
  7  >> 1 = 3
```

Limitation

In compile-time expressions, this Polyspace option does not change the standard behavior for right shifts.

For example, consider this right shift expression:

```
int arr[ ((-4) >> 20) ];
```

The compiler computes array sizes, so the expression `(-4) >> 20` is evaluated at compilation time. Logically, this expression is equivalent to 4095. However, arithmetically, the result is -1. This statement causes a compilation error (arrays cannot have negative size) because the standard right-shift behavior for signed integers is arithmetic.

Command-Line Information

When using the command line, arithmetic is the default computation mode. When this option is set, logical computation is performed.
Parameter: `-logical-signed-right-shift`
Default: Arithmetic signed right shifts
Example (Bug Finder): `polyspace-bug-finder -logical-signed-right-shift`
Example (Code Prover): `polyspace-code-prover -logical-signed-right-shift`
Example (Bug Finder Server): `polyspace-bug-finder-server -logical-signed-right-shift`
Example (Code Prover Server): `polyspace-code-prover-server -logical-signed-right-shift`

See Also

Topics

“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”

Block char16/32_t types (-no-uliterals)

Disable Polyspace definitions for `char16_t` or `char32_t`

Description

Specify that the analysis must not define `char16_t` or `char32_t` types.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See “Dependencies” on page 1-50 for other options you must also enable.

Command line: Use the option `-no-uliterals`. See “Command-Line Information” on page 1-50.

Why Use This Option

If your compiler defines `char16_t` and/or `char32_t` through a `typedef` statement or by using includes, use this option to turn off the standard Polyspace definition of `char16_t` and `char32_t`.

To check if your compiler defines these types, compile this code using the compiler settings that you typically use:

```
typedef unsigned short char16_t;
typedef unsigned long char32_t;
```

If the file compiles, it means that your compiler has already defined `char16_t` and `char32_t`. Enable this Polyspace option.

Settings

☑ On

The analysis does not allow `char16_t` and `char32_t` types.

☐ Off (default)

The analysis allows `char16_t` and `char32_t` types.

Dependencies

You can select this option only when these conditions are true:

- `Source code language (-lang)` is set to `CPP` or `C-CPP`.
- `Compiler (-compiler)` is set to `generic` or a `gnu` version.

Command-Line Information

Parameter: `-no-uliterals`
Default: off
Example (Bug Finder): `polyspace-bug-finder -lang cpp -compiler gnu4.7 -cpp-version cpp11 -no-uliterals`

Example (Code Prover): `polyspace-code-prover -compiler gnu4.7 -lang cpp -cpp-version cpp11 -no-uliterals`
Example (Bug Finder Server): `polyspace-bug-finder-server -lang cpp -compiler gnu4.7 -cpp-version cpp11 -no-uliterals`
Example (Code Prover Server): `polyspace-code-prover-server -compiler gnu4.7 -lang cpp -cpp-version cpp11 -no-uliterals`

See Also

`Compiler (-compiler)`

Topics

“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”

Pack alignment value (-pack-alignment-value)

Specify default structure packing alignment for code developed in Visual C++

Description

Specify the default packing alignment (in bytes) for structures, unions, and class members.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-pack-alignment-value`. See “Command-Line Information” on page 1-52.

Why Use This Option

If you use compiler options to specify how members of a structure are packed into memory, use this option to emulate your compiler.

For instance, if you use the Visual Studio option /Zp to specify an alignment, use this option for your Polyspace analysis.

If you use `#pragma pack` directives in your code to specify alignment, and also specify this option for analysis, the `#pragma pack` directives take precedence.

Settings

Default: 8

You can enter one of these values:

- 1
- 2
- 4
- 8
- 16

Command-Line Information

Parameter: `-pack-alignment-value`
Value: `1 | 2 | 4 | 8 | 16`
Default: 8
Example (Bug Finder): `polyspace-bug-finder -compiler visual10 -pack-alignment-value 4`
Example (Code Prover): `polyspace-code-prover -compiler visual10 -pack-alignment-value 4`
Example (Bug Finder Server): `polyspace-bug-finder-server -compiler visual10 -pack-alignment-value 4`

Example (Code Prover Server): `polyspace-code-prover-server -compiler visual10 -pack-alignment-value 4`

See Also

Ignore pragma pack directives (`-ignore-pragma-pack`)

Ignore `#pragma pack` directives

Description

Specify that the analysis must ignore `#pragma pack` directives in the code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-ignore-pragma-pack`. See “Command-Line Information” on page 1-54.

Why Use This Option

Use this option if `#pragma pack` directives in your code cause linking errors.

For instance, you have two structures with the same name in your code, but one declaration follows a `#pragma pack(2)` statement. Because the default alignment is 8 bytes, the different packing for the two structures causes a linking error. Use this option to avoid such errors.

Settings

☑ On

The analysis ignores the `#pragma` directives.

☐ Off (default)

The analysis takes into account specifications in the `#pragma` directives.

Command-Line Information

Parameter: `-ignore-pragma-pack`
Default: Off
Example (Bug Finder): `polyspace-bug-finder -ignore-pragma-pack`
Example (Code Prover): `polyspace-code-prover -ignore-pragma-pack`
Example (Bug Finder Server): `polyspace-bug-finder-server -ignore-pragma-pack`
Example (Code Prover Server): `polyspace-code-prover-server -ignore-pragma-pack`

See Also

Management of size_t (-size-t-type-is)

Specify the underlying data type of `size_t`

Description

Specify the underlying data type of `size_t` explicitly: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` or `unsigned long long`. If you do not specify this option, your choice of compiler determines the underlying type.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option `-size-t-type-is`. See “Command-Line Information” on page 1-56.

Why Use This Option

The analysis associates a data type with `size_t` when you specify your compiler. If you use a compiler option that changes this default type, emulate your compiler option by using this analysis option.

If you run into compilation errors during Polyspace analysis and trace the error to the definition of `size_t`, it is possible that you use a compiler option and change your compiler default. To probe further, compile this code with your compiler using the options that you typically use:

```
/* Header defines malloc as void* malloc (size_t size)
#include <stdio.h>

void* malloc (unsigned int size);
```

If the file does not compile, your compiler (along with compiler options) defines `size_t` using a different underlying type. Replace `unsigned int` with another type such as `unsigned long` and try again.

Settings

Default: `defined-by-compiler`

`defined-by-compiler`

Your specification for `Compiler (-compiler)` determines the underlying type of `size_t`.

`unsigned-int`

The analysis considers `unsigned int` as the underlying type of `size_t`.

`unsigned-long`

The analysis considers `unsigned long` as the underlying type of `size_t`.

`unsigned-long-long`

The analysis considers `unsigned long long` as the underlying type of `size_t`.

Command-Line Information

Parameter: `-size-t-type-is`
Value: `defined-by-compiler` | `unsigned-char` | `unsigned-int` | `unsigned-short` | `unsigned-long` | `unsigned-long-long`
Default: `defined-by-compiler`
Example (Bug Finder): `polyspace-bug-finder -size-t-type-is unsigned-long`
Example (Code Prover): `polyspace-code-prover -size-t-type-is unsigned-long`
Example (Bug Finder Server): `polyspace-bug-finder-server -size-t-type-is unsigned-long`
Example (Code Prover Server): `polyspace-code-prover-server -size-t-type-is unsigned-long`

See Also

`-custom-target`

Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

Management of wchar_t (-wchar-t-type-is)

Specify the underlying data type of wchar_t

Description

Specify the underlying data type of wchar_t explicitly. If you do not specify this option, your choice of compiler determines the underlying type.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

Command line: Use the option -wchar-t-type-is. See “Command-Line Information” on page 1-57.

Why Use This Option

The analysis associates a data type with wchar_t when you specify your compiler. If you use a compiler option that changes this default type, emulate your compiler option by using this analysis option.

Settings

Default: defined-by-compiler

defined-by-compiler

Your specification for Compiler (-compiler) determines the underlying type of wchar_t.

signed-short

The analysis considers signed short as the underlying type of wchar_t.

unsigned-short

The analysis considers unsigned short as the underlying type of wchar_t.

signed-int

The analysis considers signed int as the underlying type of wchar_t.

unsigned-int

The analysis considers unsigned int as the underlying type of wchar_t.

signed-long

The analysis considers signed long as the underlying type of wchar_t.

unsigned-long

The analysis considers unsigned long as the underlying type of wchar_t.

Command-Line Information

Parameter: -wchar-t-type-is

Value: `defined-by-compiler | signed-short | unsigned-short | signed-int | unsigned-int | signed-long | unsigned-long`
Default: `defined-by-compiler`
Example (Bug Finder): `polyspace-bug-finder -wchar-t-type-is signed-int`
Example (Code Prover): `polyspace-code-prover -wchar-t-type-is signed-int`
Example (Bug Finder Server): `polyspace-bug-finder-server -wchar-t-type-is signed-int`
Example (Code Prover Server): `polyspace-code-prover-server -wchar-t-type-is signed-int`

See Also

Topics

“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”

Ignore link errors (-no-extern-c)

Ignore certain linking errors

Description

Specify that the analysis must ignore certain linking errors.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Environment Settings** node. See “Dependency” on page 1-59 for other options that you must also enable.

Command line: Use the option `-no-extern-C`. See “Command-Line Information” on page 1-59.

Why Use This Option

Some functions may be declared inside an `extern "C" { }` block in some files and not in others. Then, their linkage is not the same and it causes a link error according to the ANSI standard.

Applying this option will cause Polyspace to ignore this error. This permissive option may not resolve all the extern C linkage errors.

Settings

☑ On

Ignore linking errors if possible.

☐ Off (default)

Stop analysis for linkage errors.

Dependency

This option is available only if you set `Source code language (-lang)` to `CPP` or `C-CPP`.

Command-Line Information

Parameter: `-no-extern-C`
Default: off
Example (Bug Finder): `polyspace-bug-finder -lang cpp -no-extern-C`
Example (Code Prover): `polyspace-code-prover -lang cpp -no-extern-C`
Example (Bug Finder Server): `polyspace-bug-finder-server -lang cpp -no-extern-C`
Example (Code Prover Server): `polyspace-code-prover-server -lang cpp -no-extern-C`

See Also

Topics

“Specify Polyspace Analysis Options”

Preprocessor definitions (-D)

Replace macros in preprocessed code

Description

Replace macros with their definitions in preprocessed code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Macros** node.

Command line: Use the option `-D`. See “Command-Line Information” on page 1-61.

Why Use This Option

Use this option to emulate your compiler behavior. For instance, if your compiler considers a macro `_WIN32` as defined when you build your code, it executes code in a `#ifdef _WIN32` statement. If Polyspace does not consider that macro as defined, you must use this option to replace the macro with 1.

Depending on your settings for `Compiler (-compiler)`, some macros are defined by default. Use this option to define macros that are not implicitly defined.

Typically, you recognize from compilation errors that a certain macro is not defined. For instance, the following code does not compile if the macro `_WIN32` is not defined.

```
#ifdef _WIN32
  int env_var;
#endif

void set() {
  env_var=1;
}
```

The error message states that `env_var` is undefined. However, the definition of `env_var` is in the `#ifdef _WIN32` statement. The underlying cause for the error is that the macro `_WIN32` is not defined. You must define `_WIN32`.

Settings

No Default

Using the  button, add a row for the macro you want to define. The definition must be in the format *`Macro`*`=`*`Value`*. If you want Polyspace to ignore the macro, leave the *`Value`* blank.

For example:

- `name1=name2` replaces all instances of `name1` by `name2`.
- `name=` instructs the software to ignore `name`.

- `name` with no equals sign or value replaces all instances of `name` by 1. To define a macro to execute code in a `#ifdef` *`macro_name`* statement, use this syntax.

Tips

- If Polyspace does not support a non-ANSI keyword and shows a compilation error, use this option to replace all occurrences of the keyword with a blank string in preprocessed code. The replacement occurs only for the purposes of the analysis. Your original source code remains intact.

  For instance, if your compiler supports the `__far` keyword, to avoid compilation errors:

  - In the user interface (desktop products only), enter `__far=`.
  - On the command line, use the flag `-D __far=`.

  The software replaces the `__far` keyword with a blank string during preprocessing. For example:

  ```
  int __far* pValue;
  ```

  is converted to:

  ```
  int * pValue;
  ```
- Polyspace recognizes keywords such as `restrict` and does not allow their use as identifiers. If you use those keywords as identifiers (because your compiler does not recognize them as keywords), replace the disallowed name with another name using this option. The replacement occurs only for the purposes of the analysis. Your original source code remains intact.

  For instance, to allow use of `restrict` as identifier:

  - In the user interface, enter `restrict=my_restrict`.
  - On the command line, use the flag `-D restrict=my_restrict`.
- Your compiler specification determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

  - To override the macro definition coming from a compiler specification, use this option.
  - To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

Command-Line Information

You can specify only one flag with each `-D` option. However, you can specify the option multiple times.
Parameter: `-D`
No Default
Value: *`flag=value`*
Example (Bug Finder): `polyspace-bug-finder -D HAVE_MYLIB -D int32_t=int`
Example (Code Prover): `polyspace-code-prover -D HAVE_MYLIB -D int32_t=int`
Example (Bug Finder Server): `polyspace-bug-finder-server -D HAVE_MYLIB -D int32_t=int`
Example (Code Prover Server): `polyspace-code-prover-server -D HAVE_MYLIB -D int32_t=int`

See Also

`Disabled preprocessor definitions (-U)`

Topics
“Specify Polyspace Analysis Options”

Disabled preprocessor definitions (-U)

Undefine macros in preprocessed code

Description

Undefine macros in preprocessed code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Macros** node.

Command line: Use the option `-U`. See “Command-Line Information” on page 1-64.

Why Use This Option

Use this option to emulate your compiler behavior. For instance, if your compiler considers a macro `_WIN32` as undefined when you build your code, it executes code in a `#ifndef _WIN32` statement. If Polyspace considers that macro as defined, you must explicitly undefine the macro.

Some settings for `Compiler (-compiler)` enable certain macros by default. This option allows you undefine the macros.

Typically, you recognize from compilation errors that a certain macro must be undefined. For instance, the following code does not compile if the macro `_WIN32` is defined.

```
#ifndef _WIN32
  int env_var;
#endif

void set() {
  env_var=1;
}
```

The error message states that `env_var` is undefined. However, the definition of `env_var` is in the `#ifndef _WIN32` statement. The underlying cause for the error is that the macro `_WIN32` is defined. You must undefine `_WIN32`.

Settings

No Default

Using the  button, add a new row for each macro being undefined.

Tips

Your compiler specification determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override a macro definition coming from a compiler specification, use the option `Preprocessor definitions (-D)`.

- To undefine the macro, use this option.

Command-Line Information

You can specify only one flag with each `-U` option. However, you can specify the option multiple times.
Parameter: `-U`
No Default
Value: *`macro`*
Example (Bug Finder): `polyspace-bug-finder -U HAVE_MYLIB -U USE_COM1`
Example (Code Prover): `polyspace-code-prover -U HAVE_MYLIB -U USE_COM1`
Example (Bug Finder Server): `polyspace-bug-finder-server -U HAVE_MYLIB -U USE_COM1`
Example (Code Prover Server): `polyspace-code-prover-server -U HAVE_MYLIB -U USE_COM1`

See Also

`Preprocessor definitions (-D)`

Topics
“Specify Polyspace Analysis Options”

Source code encoding (-sources-encoding)

Specify the encoding that the analysis uses to interpret non-ASCII characters in source code

Description

Specify the encoding of your source files. The analysis uses this information to interpret non-ASCII characters in your source code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Environment Settings** node.

Command line: Use the option `-sources-encoding`. See “Command-Line Information” on page 1-66.

Why Use This Option

If your source code contains non-ASCII characters, for instance, Japanese or Korean characters, the Polyspace analysis can interpret the characters and later display the source code correctly.

If you still have compilation errors or display issues from non-ASCII characters, you can explicitly specify your source code encoding using this option.

Settings

Default: `system`

`system`

The analysis uses the default encoding of the operating system.

`shift-jis`

The analysis uses the Shift JIS (Shift Japanese Industrial Standards) encoding, a character encoding for the Japanese language.

`iso-8859-1`

The analysis uses the ISO/IEC 8859-1:1998 encoding, a character encoding that encodes what it refers to as "Latin alphabet no.1", consisting of 191 characters from the Latin script.

`windows-1252`

The analysis uses the Windows-1252 encoding, a single-byte character encoding of the Latin alphabet, used by default in the legacy components of Windows for English and some other Western languages.

`UTF-8`

The analysis uses the UTF-8 encoding, a variable width character encoding capable of encoding all valid code points in Unicode.

Polyspace supports many more encodings. To specify an encoding that is not in the above list in the Polyspace user interface, enter `-sources-encoding` *`encodingname`* in the `Other` field. In particular, if your source files contain a mix of different encodings, you can use `-sources-encoding`

`auto`. In this mode, the analysis uses internal heuristics to determine the encoding of your source files from their contents.

For the full list of supported encodings, at the command line, enter:

```
-list-all-values -sources-encoding
```

with the `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server` command. Pipe the output to a file and search the file for the encoding that you are using.

Command-Line Information

Parameter: `-sources-encoding`
Default: `system`
Value: `auto` | `system` | `shift-jis` | `iso-8859-1` | `windows-1252` | `UTF-8`
Example (Bug Finder): `polyspace-bug-finder -sources-encoding windows-1252`
Example (Code Prover): `polyspace-code-prover -sources-encoding windows-1252`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources-encoding windows-1252`
Example (Code Prover Server): `polyspace-code-prover-server -sources-encoding windows-1252`

Polyspace supports many more encodings besides the above list. For the full list of supported encodings, at the command line, enter:

```
-list-all-values -sources-encoding
```

with the `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server` command. Pipe the output to a file and search the file for the encoding that you are using.

See Also

Topics
“Specify Polyspace Analysis Options”

Code from DOS or Windows file system (-dos)

Consider that file paths are in MS-DOS style

Description

Specify that DOS or Windows files are provided for analysis.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Environment Settings** node.

Command line: Use the option `-dos`. See “Command-Line Information” on page 1-67.

Why Use This Option

Use this option if the contents of the **Include** or **Source** folder come from a DOS or Windows file system. The option helps you resolve case sensitivity and control character issues.

Settings

☑ On (default)

Analysis understands file names and include paths for Windows/DOS files

For example, with this option,

```
#include "..\mY_TEst.h"^M

#include "..\mY_other_FILE.H"^M
```

resolves to:

```
#include "../my_test.h"

#include "../my_other_file.h"
```

In this mode, you see an error if your include folder has header files whose names differ only in case.

☐ Off

Characters are not controlled for files names or paths.

Command-Line Information

Parameter: `-dos`
Default: Off
Example (Bug Finder): `polyspace-bug-finder -dos -I ./my_copied_include_dir -D test=1`
Example (Code Prover): `polyspace-code-prover -dos -I ./my_copied_include_dir -D test=1`

Example (Bug Finder Server): `polyspace-bug-finder-server -dos -I ./my_copied_include_dir -D test=1`
Example (Code Prover Server): `polyspace-code-prover-server -dos -I ./my_copied_include_dir -D test=1`

See Also

Topics
“Specify Polyspace Analysis Options”

Stop analysis if a file does not compile (-stop-if-compile-error)

Specify that a compilation error must stop the analysis

Description

Specify that even a single compilation error must stop the analysis.

Set Option

User interface (desktop products only): In the **Configuration** pane, the option is on the **Environment Settings** node.

Command line: Use the option `-stop-if-compile-error`. See “Command-Line Information” on page 1-70.

Why Use This Option

Use this option to first resolve all compilation errors and then perform the Polyspace analysis. This sequence ensures that all files are analyzed.

Otherwise, only files without compilation errors are fully analyzed. The analysis might return some results for files that do not compile. If a file with compilation errors contains a function definition, the analysis considers the function undefined. This assumption can sometimes make the analysis less precise.

The option is more useful for a Code Prover analysis because the Code Prover run-time checks rely more heavily on range propagation across functions.

Settings

☑ On

The analysis stops even if a single compilation error occurs.

In the user interface of the Polyspace desktop products, you see the compilation errors on the **Output Summary** pane.

Type	Message	File	Line	Col
ⓘ	C verification starts at Thu Dec 17 22:26:17 2015			
ⓘ	6 core(s) detected but the verification uses 4 core(s).			
⊗	identifier "x" is undefined	my_file.c	1	
⚠	Failed compilation.	my_file.c		
⊗	Verifier has detected compilation error(s) in the code.			
⊗	Exiting because of previous error			

For information on how to resolve the errors, see “Troubleshoot Compilation Errors”.

You can also see the errors in the analysis log, a text file generated during the analysis. The log is named `Polyspace_R20##n_ProjectName_date-time.log` and contains lines starting with `Error:` indicating compilation errors. To view the log from the analysis results:

- In the user interface of the Polyspace desktop products, select **Window > Show/Hide View > Run Log**.
- In the Polyspace Access web interface, open the **Review** tab. Select **Layout > Show/Hide View > Run Log**.

Despite compilation errors, you can see some analysis results, for instance, coding rule violations.

☐ Off (default)

The analysis does not stop because of compilation errors, but only files without compilation errors are analyzed. The analysis does not consider files that do not compile. If a file with compilation errors contains a function definition, the analysis considers the function undefined. If the analysis needs the definition of such a function, it makes broad assumptions about the function.

- The function return value can take any value in the range allowed by its data type.
- The function can modify arguments passed by reference so that they can take any value in the range allowed by their data types.

If the assumptions are too broad, the analysis can be less precise. For instance, a run-time check can flag an operation in orange even though it does not fail in practice.

If compilation errors occur, in the user interface of the Polyspace desktop products, the **Dashboard** pane has a link, which shows that some files failed to compile. You can click the link and see the compilation errors on the **Output Summary** pane.

You can also see the errors in the analysis log, a text file generated during the analysis. The log is named `Polyspace_R20##n_ProjectName_date-time.log` and contains lines starting with `Error:` indicating compilation errors. To view the log from the analysis results:

- In the user interface of the Polyspace desktop products, select **Window > Show/Hide View > Run Log**.
- In the Polyspace Access web interface, open the **Review** tab. Select **Layout > Show/Hide View > Run Log**.

Command-Line Information

Parameter: `-stop-if-compile-error`
Default: Off
Example (Bug Finder): `polyspace-bug-finder -sources filename -stop-if-compile-error`
Example (Code Prover): `polyspace-code-prover -sources filename -stop-if-compile-error`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources filename -stop-if-compile-error`
Example (Code Prover Server): `polyspace-code-prover-server -sources filename -stop-if-compile-error`

See Also

Topics
“Specify Polyspace Analysis Options”

Introduced in R2017a

Command/script to apply to preprocessed files (-post-preprocessing-command)

Specify command or script to run on source files after preprocessing phase of analysis

Description

Specify a command or script to run on each source file after preprocessing.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Environment Settings** node.

Command line: Use the option `-post-preprocessing-command`. See “Command-Line Information” on page 1-74.

Why Use This Option

You can run scripts on preprocessed files to work around compilation errors or imprecisions of the analysis while keeping your original source files untouched. For instance, suppose Polyspace does not recognize a compiler-specific keyword. If you are certain that the keyword is not relevant for the analysis, you can run a Perl script to remove all instances of the keyword. When you use this option, the software removes the keyword from your preprocessed code but keeps your original code untouched.

Use a script only if the existing analysis options do not meet your requirements. For instance:

- For direct replacement of one keyword with another, use the option `Preprocessor definitions (-D)`.

  However, the option does not allow search and replacement involving regular expressions. For regular expressions, use a script.
- For mapping your library function to a standard library function, use the option `-code-behavior-specifications`.

  However, the option supports mapping to only a subset of standard library functions. To map to an unsupported function, use a script.

If you are unsure about removing or replacing an unsupported construct, do not use this option. Contact MathWorks® Support for guidance.

Settings

No Default

Enter full path to the command or script or click  to navigate to the location of the command or script. This script is executed before verification.

Tips

- Your script must be designed to process the standard output from preprocessing and produce its results in accordance with that standard output.
- Your script must preserve the number of lines in the preprocessed file. In other words, it must not add or remove entire lines to or from the file.

  Adding a line or removing one can potentially result in some unpredictable behavior on the location of checks and macros in the Polyspace user interface.
- For a Perl script, in Windows, specify the full path to the Perl executable followed by the full path to the script.

  For example:

  - To specify a Perl command that replaces all instances of the `far` keyword, enter *`polyspaceroot`*`\sys\perl\win32\bin\perl.exe -p -e "s/far//g"`.
  - To specify a Perl script `replace_keyword.pl` that replaces all instances of a keyword, enter *`polyspaceroot`*`\sys\perl\win32\bin\perl.exe` *`absolute_path`*`\replace_keyword.pl`.

  Here, *`polyspaceroot`* is the location of the current Polyspace installation such as `C:\Program Files\Polyspace\R2019a\` and *`absolute_path`* is the location of the Perl script. If the paths contain spaces, use quotes to enclose the full path names.
- Use this Perl script as template. The script removes all instances of the `far` keyword.

```
#!/usr/bin/perl

binmode STDOUT;

# Process every line from STDIN until EOF
while ($line = <STDIN>)
{

  # Remove far keyword
  $line =~ s/far//g;

  # Print the current processed line to STDOUT
  print $line;
}
```

  You can use Perl regular expressions to perform substitutions. For instance, you can use the following expressions.

Expression	Meaning
.	Matches any single character except newline
[a-z0-9]	Matches any single letter in the set a-z, or digit in the set 0-9
[^a-e]	Matches any single letter not in the set a-e
\d	Matches any single digit
\w	Matches any single alphanumeric character or _
x?	Matches 0 or 1 occurrence of x

Expression	Meaning
x*	Matches 0 or more occurrences of x
x+	Matches 1 or more occurrences of x

For complete list of regular expressions, see Perl documentation.

- When you specify this option, the Compilation Assistant is automatically disabled.

Command-Line Information

Parameter: `-post-preprocessing-command`
Value: Path to executable file or command in quotes
No Default
Example in Linux® (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-post-preprocessing-command` `` `pwd`/replace_keyword.pl ``
Example in Linux (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-post-preprocessing-command` `` `pwd`/replace_keyword.pl ``
Example in Linux (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-post-preprocessing-command` `` `pwd`/replace_keyword.pl ``
Example in Linux (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-post-preprocessing-command` `` `pwd`/replace_keyword.pl ``
Example in Windows: `polyspace-bug-finder -sources` *`file_name`* `-post-preprocessing-command "C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\replace_keyword.pl"`

Note that in Windows, you use the full path to the Perl executable.

See Also

`-regex-replace-rgx -regex-replace-fmt` | `Command/script to apply after the end of the code verification (-post-analysis-command)`

Topics

“Specify Polyspace Analysis Options”
“Remove or Replace Keywords Before Compilation”

Include (-include)

Specify files to be #include-ed by each C file in analysis

Description

Specify files to be #include-ed by each C file involved in the analysis. The software enters the #include statements in the preprocessed code used for analysis, but does not modify the original source code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Environment Settings** node.

Command line: Use the option -include. See “Command-Line Information” on page 1-75.

Why Use This Option

There can be many reasons why you want to #include a file in all your source files.

For instance, you can collect in one header file all workarounds for compilation errors. Use this option to provide the header file for analysis. Suppose you have compilation issues because Polyspace does not recognize certain compiler-specific keywords. To work around the issues, #define the keywords in a header file and provide the header file with this option.

Settings

No Default

Specify the file name to be included in every file involved in the analysis.

Polyspace still acts on other directives such as #include <include_file.h>.

Command-Line Information

Parameter: -include
Default: None
Value: *file* (Use -include multiple times for multiple files)
Example (Bug Finder): polyspace-bug-finder -include `pwd`/sources/a_file.h -include /inc/inc_file.h
Example (Code Prover): polyspace-code-prover -include `pwd`/sources/a_file.h -include /inc/inc_file.h
Example (Bug Finder Server): polyspace-bug-finder-server -include `pwd`/sources/a_file.h -include /inc/inc_file.h
Example (Code Prover Server): polyspace-code-prover-server -include `pwd`/sources/a_file.h -include /inc/inc_file.h

See Also

Topics

“Specify Polyspace Analysis Options”

“Gather Compilation Options Efficiently”

Include folders (-I)

View include folders used for analysis

Description

This option is relevant only for the user interface of the Polyspace desktop products.

View the include folders used for analysis.

Set Option

This is not an option that you set in your project configuration. You can only view the include folders in the configuration associated with a result. For instance, in the user interface:

- To add include folders, on the **Project Browser**, right-click your project. Select **Add Source**.
- To view the include folders that you used, with your results open, select **Window > Show/Hide View > Configuration**. Under the node **Environment Settings**, you see the folders listed under **Include folders**.

Settings

This is a read-only option available only when viewing results in the user interface of the Polyspace desktop products. Unlike other options, you do not specify include folders on the **Configuration** pane. Instead, you add your include folders on the **Project Browser** pane.

See Also

`-I` | `Include (-include)`

Constraint setup (-data-range-specifications)

Constrain global variables, function inputs and return values of stubbed functions

Description

This option applies primarily to a Code Prover analysis. In Bug Finder, you can only specify external constraints on global variables.

Specify constraints (also known as data range specifications or DRS) for global variables, function inputs and return values of stubbed functions using a **Constraint Specification** template file. The template file is an `XML` file that you can generate in the Polyspace user interface.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-data-range-specifications`. See “Command-Line Information” on page 1-79.

Why Use This Option

Use this option for specifying constraints outside your code.

Polyspace uses the code that you provide to make assumptions about items such as variable ranges and allowed buffer size for pointers. Sometimes the assumptions are broader than what you expect because:

- You have not provided the complete code. For example, you did not provide some of the function definitions.
- Some of the information about variables is available only at run time. For example, some variables in your code obtain values from the user at run time.

Because of these broad assumptions:

- Code Prover can consider more execution paths than those paths that occur at run time. If an operation fails along one of the execution paths, Polyspace places an orange check on the operation. If that execution path does not occur at run time, the orange check indicates a false positive.
- Bug Finder can sometimes produce false positives.

To reduce the number of such false positives, you can specify additional constraints on global variables, function inputs, and return values of stubbed functions.

After you specify your constraints, you can save them as an XML file to use them for subsequent analyses. If your source code changes, you can update the previous constraints. You do not have to create a new constraint template.

Settings

No Default

Enter full path to the template file. Alternately, click 

to open a **Constraint Specification** wizard. This wizard allows you to generate a template file or navigate to an existing template file.

For more information, see “Specify External Constraints”.

Command-Line Information

Parameter: `-data-range-specifications`
Value: *`file`*
No Default
Example (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-data-range-specifications "C:\DRS\range.xml"`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-data-range-specifications "C:\DRS\range.xml"`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-data-range-specifications "C:\DRS\range.xml"`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-data-range-specifications "C:\DRS\range.xml"`

See Also

`Functions to stub (-functions-to-stub)` | `Ignore default initialization of global variables (-no-def-init-glob)`

Topics

“Specify Polyspace Analysis Options”
“Specify External Constraints”

Ignore default initialization of global variables (-no-def-init-glob)

Consider global variables as uninitialized unless explicitly initialized in code

Description

This option applies to Code Prover only. It does not affect a Bug Finder analysis.

Specify that Polyspace must not consider global and static variables as initialized unless they are explicitly initialized in the code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-no-def-init-glob`. See “Command-Line Information” on page 1-81.

Why Use This Option

The C99 Standard specifies that global variables are implicitly initialized. The default analysis follows the Standard and considers this implicit initialization.

If you want to initialize specific global variables explicitly, use this option to find the instances where global variables are not explicitly initialized.

Settings

☑ On

Polyspace ignores implicit initialization of global and static variables. The verification generates a red **Non-initialized variable** error if your code reads a global or static variable before writing to it.

If you enable this option, global variables are considered uninitialized unless you explicitly initialize them in the code. Note that this option overrides the option `Variables to initialize (-main-generator-writes-variables)`. Even if you initialize variables with the generated `main`, this option forces the analysis to ignore the initialization.

☐ Off (default)

Polyspace considers global variables and static variables to be initialized according to C99 or ISO C++ standards. For instance, the default values are:

- 0 for `int`
- 0 for `char`
- 0.0 for `float`

Tips

Static local variables have the same lifetime as global variables even though their visibility is limited to the function where they are defined. Therefore, the option applies to static local variables.

Command-Line Information

Parameter: `-no-def-init-glob`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-no-def-init-glob`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-no-def-init-glob`

See Also

`Non-initialized variable`

Topics
“Specify Polyspace Analysis Options”

No STL stubs (`-no-stl-stubs`)

Do not use Polyspace implementations of functions in the Standard Template Library

Description

Specify that the verification must not use Polyspace implementations of the Standard Template Library.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node. See “Dependency” on page 1-82 for other options that you must also enable.

Command line: Use the option `-no-stl-stubs`. See “Command-Line Information” on page 1-82.

Why Use This Option

The analysis uses an efficient implementation of all class templates from the Standard Template Library (STL). If your compiler redefines the templates, in some cases, your compiler implementation can conflict with the Polyspace implementation.

Use this option to prevent Polyspace from using its implementations of STL templates. You must also explicitly provide the path to your compiler includes. See “C++ Standard Template Library Stubbing Errors” (Polyspace Code Prover).

Settings

☑ On

The verification does not use Polyspace implementations of the Standard Template Library.

☐ Off (default)

The verification uses efficient Polyspace implementations of the Standard Template Library.

Dependency

This option is available only if you set `Source code language (-lang)` to `CPP` or `C-CPP`.

Command-Line Information

Parameter: `-no-stl-stubs`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-no-stl-stubs`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-no-stl-stubs`

See Also

Topics
“Specify Polyspace Analysis Options”

Functions to stub (`-functions-to-stub`)

Specify functions to stub during analysis

Description

Specify functions to stub during analysis.

For specified functions, Polyspace :

- Ignores the function definition even if it exists.
- Assumes that the function inputs and outputs have full range of values allowed by their type.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-functions-to-stub`. See “Command-Line Information” on page 1-85.

Why Use This Option

If you want the analysis to ignore the code in a function body, you can stub the function.

For instance:

- Suppose you have not completed writing the function and do not want the analysis to consider the function body. You can use this option to stub the function and then specify constraints on its return value and modifiable arguments.
- Suppose the analysis of a function body is imprecise. The analysis assumes that the function returns all possible values that the function return type allows. You can use this option to stub the function and then specify constraints on its return value.

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

When entering function names, use either the basic syntax or, to differentiate overloaded functions, the argument syntax. For the argument syntax, separate function arguments with semicolons. See the following code and table for examples.

```
//simple function

void test(int a, int b);
```

```
//C++ template function

Template <class myType>
myType test(myType a, myType b);

//C++ class method

class A {
    public:
    int test(int var1, int var2);
};


//C++ template class method

template <class myType> class A
{
    public:
    myType test(myType var1, myType var2);
};
```

Function Type	Basic Syntax	Argument Syntax
Simple function	`test`	`test(int; int)`
C++ template function	`test`	`test(myType; myType)`
C++ class method	`A::test`	`A::test(int;int)`
C++ template class method	`A<myType>::test`	`A<myType>::test(myType;myType)`

Tips

- Code Prover makes assumptions about the arguments and return values of stubbed functions. For example, Polyspace assumes that the return values of stubbed functions are full range. These assumptions can affect checks in other sections of the code. See “Stubbed Functions” (Polyspace Code Prover).
- If you stub a function, you can constrain the range of function arguments and return value. To specify constraints, use the analysis option `Constraint setup (-data-range-specifications)`.
- For C functions, these special characters are allowed: `( ) < > ; _`

  For C++ functions, these special characters are allowed : `( ) < > ; _ * & [ ]`

  Space characters are allowed for C++, but are not allowed for C functions.

Command-Line Information

Parameter: `-functions-to-stub`
No Default
Value: *`function1`*`[,`*`function2`*`[,...]]`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-functions-to-stub function_1,function_2`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -functions-to-stub function_1,function_2`

See Also

`Constraint setup (-data-range-specifications)`

Topics

“Specify Polyspace Analysis Options”

Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions)

Stub autogenerated functions that use lookup tables and model them more precisely

Description

This option is available only for model-generated code. The option is relevant only if you generate code from a Simulink® model that uses Lookup Table blocks using MathWorks code generation products.

Specify that the verification must stub autogenerated functions that use certain kinds of lookup tables in their body. The lookup tables in these functions use linear interpolation and do not allow extrapolation. That is, the result of using the lookup table always lies between the lower and upper bounds of the table.

Set Option

If you are running verification from Simulink, use the option “Stub lookup tables” (Polyspace Code Prover) in Simulink Configuration Parameters, which performs the same task.

User interface (desktop products only): In your Polyspace project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-stub-embedded-coder-lookup-table-functions`. See “Command-Line Information” on page 1-88.

Why Use This Option

If you use this option, the verification is more precise and has fewer orange checks. The verification of lookup table functions is usually imprecise. The software has to make certain assumptions about these functions. To avoid missing a run-time error, the verification assumes that the result of using the lookup table is within the full range allowed by the result data type. This assumption can cause many unproven results (orange checks) when a lookup table function is called. By using this option, you narrow down the assumption. For functions that use lookup tables with linear interpolation and no extrapolation, the result is at least within the bounds of the table.

The option is relevant only if your model has Lookup Table blocks. In the generated code, the functions corresponding to Lookup Table blocks also use lookup tables. The function names follow specific conventions. The verification uses the naming conventions to identify if the lookup tables in the functions use linear interpolation and no extrapolation. The verification then replaces such functions with stubs for more precise verification.

Settings

☑ On (default)

For autogenerated functions that use lookup tables with linear interpolation and no extrapolation, the verification:

- Does not check for run-time errors in the function body.
- Calls a function stub instead of the actual function at the function call sites. The stub ensures that the result of using the lookup table is within the bounds of the table.

To identify if the lookup table in the function uses linear interpolation and no extrapolation, the verification uses the function name. In your analysis results, you see that the function is not analyzed. If you place your cursor on the function name, you see the following message:

```
Function has been recognized as an Embedded Coder Lookup-Table function.
It was stubbed by Polyspace to increase precision.
Unset the -stub-embedded-coder-lookup-table-functions option to analyze
   the code below.
```

Off

The verification does not stub autogenerated functions that use lookup tables.

Tips

- The option applies to only autogenerated functions. If you integrate your own C/C++ S-Function using lookup tables with the model, these functions do not follow the naming conventions for autogenerated functions. The option does not cause them to be stubbed. If you want the same behavior for your handwritten lookup table functions as the autogenerated functions, use the option `-code-behavior-specifications` and map your function to the `__ps_lookup_table_clip` function.
- If you run verification from Simulink, the option is on by default. For certification purposes, if you want your verification tool to be independent of the code generation tool, turn off the option.

Command-Line Information

Parameter: `-stub-embedded-coder-lookup-table-functions`
Default: On
Example (Code Prover): `polyspace-code-prover -sources file_name -stub-embedded-coder-lookup-table-functions`
Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -stub-embedded-coder-lookup-table-functions`

See Also

Topics
“Specify Polyspace Analysis Options”

Introduced in R2016b

Generate results for sources and (-generate-results-for)

Specify files on which you want analysis results

Description

Specify files on which you want analysis results.

The option applies only to coding rule violations and code metrics. You cannot suppress Code Prover run-time checks from select source and header files.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-generate-results-for`. See “Command-Line Information” on page 1-90.

Why Use This Option

Use this option to see results in header files that are most relevant to you.

For instance, by default, results are generated on header files that are located in the same folder as the source files. Often, other header files belong to a third-party library. Though these header files are required for a precise analysis, you are not interested in reviewing findings in those headers. Therefore, by default, results are not generated for those headers. If you *are interested* in certain headers from third-party libraries, change the default value of this option.

Settings

Default: `source-headers`

`source-headers`

Results appear on source files and header files in the same folder as the source files or in subfolders of source file folders.

The source files are the files that you add to the **Source** folder of your Polyspace project (or use with the argument `-sources` at the command line).

`all-headers`

Results appear on source files and all header files. The header files can be in the same folder as source files, in subfolders of source file folders or in include folders.

The source files are the files that you add to the **Source** folder of your Polyspace project (or use with the argument `-sources` at the command line).

The include folders are the folders that you add to the **Include** folder of your Polyspace project (or use with the argument `-I` at the command line).

`custom`

Results appear on source files and the files that you specify. If you enter a folder name, results appear on header files in that folder.

Click  to add a field. Enter a file or folder name.

Tips

1. Use this option in combination with appropriate values for the option `Do not generate results for (-do-not-generate-results-for)`.

   If you choose `custom` and the values for the two options conflict, the more specific value determines the display of results. For instance, in the following examples, the value for the option **Generate results for sources and** is more specific.

Generate results for sources and	Do not generate results for	Final Result
`custom`: `C:\Includes\Custom_Library\`	`custom`: `C:\Includes`	Results are displayed on header files in `C:\Includes\Custom_Library\` but not generated for other header files in `C:\Includes` and its subfolders.
`custom`: `C:\Includes\my_header.h`	`custom`: `C:\Includes\`	Results are displayed on the header file `my_header.h` in `C:\Includes\` but not generated for other header files in `C:\Includes\` and its subfolders.

   Using these two options together, you can suppress results from all files in a certain folder but unsuppress select files in those folders.

2. If you choose `all-headers` for this option, results are displayed on all header files irrespective of what you specify for the option **Do not generate results for**.

Command-Line Information

Parameter: `-generate-results-for`
Value: `source-headers` | `all-headers` | `custom=`*`file1`*`[,`*`file2`*`[,...]]` | `custom=`*`folder1`*`[,`*`folder2`*`[,...]]`
Example (Bug Finder): `polyspace-bug-finder -lang c -sources` *`file_name`* `-misra2 required-rules -generate-results-for custom="C:\usr\include"`
Example (Code Prover): `polyspace-code-prover -lang c -sources` *`file_name`* `-misra2 required-rules -generate-results-for custom="C:\usr\include"`
Example (Bug Finder Server): `polyspace-bug-finder-server -lang c -sources` *`file_name`* `-misra2 required-rules -generate-results-for custom="C:\usr\include"`
Example (Code Prover Server): `polyspace-code-prover-server -lang c -sources` *`file_name`* `-misra2 required-rules -generate-results-for custom="C:\usr\include"`

See Also

Topics
“Specify Polyspace Analysis Options”

Introduced in R2016a

Do not generate results for (-do-not-generate-results-for)

Specify files on which you do not want analysis results

Description

Specify files on which you do not want analysis results.

The option applies only to coding rule violations, code metrics and unused global variables. You cannot suppress Code Prover run-time checks from source and header files.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node.

Command line: Use the option `-do-not-generate-results-for`. See “Command-Line Information” on page 1-95.

Why Use This Option

Use this option to see results in header files that are most relevant to you.

For instance, by default, results are generated on header files that are located in the same folder as the source files. If you are not interested in reviewing the findings in those headers, change the default value of this option.

Settings

Default: `include-folders`

`include-folders`

Results are not generated for header files in include folders.

The include folders are the folders that you add to the **Include** folder of your Polyspace project (or use with the argument `-I` at the command line).

If an include folder is a subfolder of a source folder, results are generated for files in that include folder even if you specify the option value `include-folders`. In this situation, use the option value `custom` and explicitly specify the include folders to ignore.

`all-headers`

Results are not generated for all header files. The header files can be in the same folder as source files, in subfolders of source file folders or in include folders.

The source files are the files that you add to the **Source** folder of your Polyspace project (or use with the argument `-sources` at the command line).

The include folders are the folders that you add to the **Include** folder of your Polyspace project (or use with the argument `-I` at the command line).

`custom`

Results are not generated for the files that you specify. If you enter a folder name, results are suppressed from files in that folder.

Click  to add a field. Enter a file or folder name.

Tips

1. Use this option appropriately in combination with appropriate values for the option `Generate results for sources and (-generate-results-for)`.

   If you choose `custom` and the values for the two options conflict, the more specific value determines the display of results. For instance, in the following examples, the value for the option **Generate results for sources and** is more specific.

Generate results for sources and	Do not generate results for	Final Result
`custom`: `C:\Includes\Custom_Library\`	`custom`: `C:\Includes`	Results are displayed on header files in `C:\Includes\Custom_Library\` but not generated for other header files in `C:\Includes` and its subfolders.
`custom`: `C:\Includes\my_header.h`	`custom`: `C:\Includes\`	Results are displayed on the header file `my_header.h` in `C:\Includes\` but not generated for other header files in `C:\Includes\` and its subfolders.

   Using these two options together, you can suppress results from all files in a certain folder but unsuppress select files in those folders.

2. If you choose `all-headers` for this option, results are suppressed from all header files irrespective of what you specify for the option **Generate results for sources and**.

3. If a defect or coding rule violation involves two files and you do not generate results for one of the files, the defect or rule violation still appears. For instance, if you define two variables with similar-looking names in files `myFile.cpp` and `myFile.h`, you get a violation of the MISRA® C++ rule 2-10-1, even if you do not generate results for `myFile.h`. MISRA C++ rule 2-10-1 states that different identifiers must be typographically unambiguous.

   The following results can involve more than one file:

   MISRA C: 2004 Rules

   - MISRA C: 2004 Rule 5.1 — Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
   - MISRA C: 2004 Rule 5.2 — Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
   - MISRA C: 2004 Rule 8.8 — An external object or function shall be declared in one file and only one file.

- MISRA C: 2004 Rule 8.9 — An identifier with external linkage shall have exactly one external definition.

MISRA C: 2012 Directives and Rules

- MISRA C: 2012 Directive 4.5 — Identifiers in the same name space with overlapping visibility should be typographically unambiguous.
- MISRA C: 2012 Rule 5.2 — Identifiers declared in the same scope and name space shall be distinct.
- MISRA C: 2012 Rule 5.3 — An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
- MISRA C: 2012 Rule 5.4 — Macro identifiers shall be distinct.
- MISRA C: 2012 Rule 5.5 — Identifiers shall be distinct from macro names.
- MISRA C: 2012 Rule 8.5 — An external object or function shall be declared once in one and only one file.
- MISRA C: 2012 Rule 8.6 — An identifier with external linkage shall have exactly one external definition.

MISRA C++ Rules

- MISRA C++ Rule 2-10-1 — Different identifiers shall be typographically unambiguous.
- MISRA C++ Rule 2-10-2 — Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
- MISRA C++ Rule 3-2-2 — The One Definition Rule shall not be violated.
- MISRA C++ Rule 3-2-3 — A type, object or function that is used in multiple translation units shall be declared in one and only one file.
- MISRA C++ Rule 3-2-4 — An identifier with external linkage shall have exactly one definition.
- MISRA C++ Rule 7-5-4 — Functions should not call themselves, either directly or indirectly.
- MISRA C++ Rule 15-4-1 — If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.

JSF C++ Rules

- JSF C++ Rule 46 — User-specified identifiers (internal and external) will not rely on significance of more than 64 characters.
- JSF C++ Rule 48 — Identifiers will not differ by only a mixture of case, the presence/absence of the underscore character, the interchange of the letter `O` with the number 0 or the letter `D`, the interchange of the letter `I` with the number 1 or the letter `l`, the interchange of the letter `S` with the number 5, the interchange of the letter `Z` with the number 2 and the interchange of the letter `n` with the letter `h`.
- JSF C++ Rule 137 — All declarations at file scope should be static where possible.
- JSF C++ Rule 139 — External objects will not be declared in more than one file.

Polyspace Bug Finder Defects

- `Variable shadowing` — Variable hides another variable of same name with nested scope.
- `Declaration mismatch` — Mismatch occurs between function or variable declarations.

4 If a global variable is never used after declaration, it appears in Code Prover results as an unused global variable. However, if it is declared in a file for which you do not want results, you do not see the unused variable in your verification results.

5 If a result (coding rule violation or Bug Finder defect) is inside a macro, Polyspace typically shows the result on the macro definition instead of the macro occurrences so that you review the result only once. Even if the macro is used in a suppressed file, the result is still shown on the macro definition, *if the definition occurs in an unsuppressed file*.

Command-Line Information

Parameter: `-do-not-generate-results-for`
Value: `all-headers` | `include-folders` | `custom=`*`file1`*`[,`*`file2`*`[,...]]` | `custom=`*`folder1`*`[,`*`folder2`*`[,...]]`
Example (Bug Finder): `polyspace-bug-finder -lang c -sources` *`file_name`* `-misra2 required-rules -do-not-generate-results-for custom="C:\usr\include"`
Example (Code Prover): `polyspace-code-prover -lang c -sources` *`file_name`* `-misra2 required-rules -do-not-generate-results-for custom="C:\usr\include"`
Example (Bug Finder Server): `polyspace-bug-finder-server -lang c -sources` *`file_name`* `-misra2 required-rules -do-not-generate-results-for custom="C:\usr\include"`
Example (Code Prover Server): `polyspace-code-prover-server -lang c -sources` *`file_name`* `-misra2 required-rules -do-not-generate-results-for custom="C:\usr\include"`

See Also

`Generate results for sources and (-generate-results-for)`

Topics
“Specify Polyspace Analysis Options”

Introduced in R2016a

External multitasking configuration

Enable setup of multitasking configuration from external file definitions

Description

This option is not available for code generated from MATLAB code or Simulink models.

Specify whether you want to use definitions from external files to set up the multitasking configuration of your Polyspace project. The supported external file formats are:

- ARXML files for AUTOSAR projects
- OIL files for OSEK projects

Set Option

User interface: In the **Configuration** pane, the option is available on the **Multitasking** node.

Command line: See “Command-Line Information” on page 1-96.

Why Use This Option

If your AUTOSAR project includes ARXML files with ECU configuration parameters, or if your OSEK project includes OIL files, Polyspace can parse these files. The software sets up tasks, interrupts, cyclical tasks, and critical sections. You do not have to set them up manually.

Settings

☑ On

Polyspace parses the external files that you provide in the format that you specify to set up the multitasking configuration of your project.

`osek`

Look for and parse OIL files to extract multitasking description.

`autosar`

Look for and parse AUTOSAR XML files to extract multitasking description.

☐ Off (default)

Polyspace does not set up the multitasking configuration of your project.

Command-Line Information

There is no single command-line option to turn on external multitasking configuration. By using the `-osek-multitasking` option or the `-autosar-multitasking` option, you enable external multitasking configuration.

See Also

`ARXML files selection (-autosar-multitasking)` | `OIL files selection (-osek-multitasking)`

Topics
“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”

Introduced in R2018a

OIL files selection (`-osek-multitasking`)

Set up multitasking configuration from OIL file definition

Description

This option is not available for code generated from MATLAB code or Simulink models.

Specify the OIL files that Polyspace parses to set up the multitasking configuration of your OSEK project.

Set Option

User interface: In the **Configuration** pane, the option is available on the **Multitasking** node. See Dependencies on page 1-101 for other options you must also enable.

Command line: Use the option `-osek-multitasking`. See "Command-Line Information" on page 1-101.

Why Use This Option

If your project includes OIL files, Polyspace can parse these files to set up tasks, interrupts, cyclical tasks, and critical sections. You do not have to set them up manually.

Settings

☑ On

Polyspace looks for and parses OIL files to set up your multitasking configuration.

`auto`

Look for OIL files in your project source and include folders, but not in their subfolders.

`custom`

Look for OIL files on the specified path and the path subfolders. You can specify a path to the OIL files or to the folder containing the files.

When you select this option, in your source code, Polyspace supports these OSEK multitasking keywords:

- `TASK`
- `DeclareTask`
- `ActivateTask`
- `DeclareResource`
- `GetResource`
- `ReleaseResource`
- `ISR`
- `DeclareEvent`
- `DeclareAlarm`

Polyspace parses the OIL files that you provide for TASK, ISR, RESOURCE, and ALARM definitions. The analysis uses these definitions and the supported multitasking keywords to configure tasks, interrupts, cyclical tasks, and critical sections.

Example: Analyze Your OSEK Multitasking Project

This example shows how to set up the multitasking configuration of an OSEK project and run an analysis on this project. To try the steps in this example, use the demo files in the folder *polyspaceroot*/help/toolbox/bugfinder/examples/External_multitasking/OSEK or *polyspaceroot*/help/toolbox/codeprover/examples/External_multitasking/OSEK. *polyspaceroot* is the Polyspace installation folder. The analysis results apply to this example code.

```
#include <assert.h>
#include "include/example_osek_multi.h"

int var1;
int var2;
int var3;

DeclareAlarm(Cyclic_task_activate);
DeclareResource(res1);
DeclareTask(init);
TASK(afterinit1);

TASK(init) // task
{


  var2++;
  ActivateTask(afterinit1);
  var3++;
  GetResource(res1); // critical section begins
  var1++;
  ReleaseResource(res1); // critical section ends
}

TASK(afterinit1) // task
{
  var3++;
  var2++;
  GetResource(res1); // critical section begins
  var1++;
  ReleaseResource(res1); // critical section ends

}


void main()
{}
```

To set up your multitasking configuration and analyze the code:

1. Copy the contents of *polyspaceroot*/help/toolbox/bugfinder/examples/External_multitasking/OSEK or *polyspaceroot*/help/toolbox/codeprover/examples/External_multitasking/OSEK to your machine, for instance in C:\Polyspace_worskpace\OSEK.

2 Run an analysis on your OSEK project by using the command:

- Bug Finder:

```
polyspace-bug-finder -sources ^
C:\Polyspace_workspace\OSEK\example_osek_multitasking.c ^
-osek-multitasking auto
```

- Code Prover:

```
polyspace-code-prover -sources ^
C:\Polyspace_workspace\OSEK\example_osek_multitasking.c ^
-osek-multitasking auto
```

- Bug Finder Server:

```
polyspace-bug-finder-server -sources ^
C:\Polyspace_workspace\OSEK\example_osek_multitasking.c ^
-osek-multitasking auto
```

- Code Prover Server:

```
polyspace-code-prover-server -sources ^
C:\Polyspace_workspace\OSEK\example_osek_multitasking.c ^
-osek-multitasking auto
```

Bug Finder detects a data race on variable `var3` because of multiple read and write operation from tasks `init` and `afterinit1`. See `Data race`.

```
#include <assert.h>
#include "include/example_osek_multi.h"

int var1;
int var2;
int var3;
```

There is no defect on `var2` since `afterinit1` goes to an active state (`ActivateTask()`) after `init` increments `var2`. Similarly, there is no defect on `var1` because it is protected by the `GetResource()` and `ReleaseResource()` calls.

Code Prover detects that `var3` is a potentially unprotected global variable because it is used in tasks `init` and `afterinit1` with no protection from interruption during the read and write operations. The analysis also shows that the cyclic task operation on `var4` can potentially cause an overflow. See `Potentially unprotected variable` and `Overflow`.

```
#include <assert.h>
#include "include/example_osek_multi.h"

int var1;
int var2;
int var3;

...
void func()
{
  var4++;
}
```

Variable var2 is not shared because afterinit1 goes to an active state (ActivateTask()) after init increments var2. Variable var1 is a protected variable through the critical sections from the GetResource() and ReleaseResource() calls.

To see how Polyspace models the TASK, ISR, and RESOURCE definitions from your OIL files, open the **Concurrency window** from the **Dashboard** pane.

Off (default)

Polyspace does not set up a multitasking configuration for your OSEK project.

Additional Considerations

- The analysis ignores TerminateTask() declarations in your source code and considers that subsequent code is executed.
- Polyspace ignores syntax elements of your OIL files that do not follow the syntax defined here.

Dependencies

To enable this option in the user interface of the desktop products, first select the option External multitasking configuration.

Command-Line Information

Parameter: -osek-multitasking
Value: auto | custom='*file1 [,file2, dir1,...]*'
Default: Off
Example (Bug Finder): polyspace-bug-finder -sources *source_path* -I *include_path* -osek-multitasking custom='path\to\file1.oil, path\to\dir'
Example (Code Prover): polyspace-code-prover -sources *source_path* -I *include_path* -osek-multitasking custom='path\to\file1.oil, path\to\dir'
Example (Bug Finder Server): polyspace-bug-finder-server -sources *source_path* -I *include_path* -osek-multitasking custom='path\to\file1.oil, path\to\dir'
Example (Code Prover Server): polyspace-code-prover-server -sources *source_path* -I *include_path* -osek-multitasking custom='path\to\file1.oil, path\to\dir'

See Also

Topics

“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”

Introduced in R2017b

ARXML files selection (-autosar-multitasking)

Set up multitasking configuration from ARXML file definitions

Description

To detect data races in large AUTOSAR applications, use this option with Polyspace Bug Finder.

This option is not available for code generated from MATLAB code or Simulink models.

Specify the ARXML files that Polyspace parses to set up the multitasking configuration of your AUTOSAR project.

Set Option

User interface: In the **Configuration** pane, the option is available on the **Multitasking** node. See Dependencies on page 1-103 for other options you must also enable.

Command line: Use the option `-autosar-multitasking`. See “Command-Line Information” on page 1-101.

Why Use This Option

If your project includes ARXML files with `<ECUC-CONTAINER-VALUE>` elements, Polyspace can parse these files to set up tasks, interrupts, cyclical tasks, and critical sections. You do not have to set them up manually.

Settings

☑ On

Polyspace looks for and parses ARXML files to set up your multitasking configuration.

When you select this option, the software assumes that you use the OSEK multitasking API in your source code to declare and define tasks and interrupts. Polyspace supports these OSEK multitasking keywords:

- `TASK`
- `DeclareTask`
- `ActivateTask`
- `DeclareResource`
- `GetResource`
- `ReleaseResource`
- `ISR`
- `DeclareEvent`
- `DeclareAlarm`

Polyspace parses the ARXML files that you provide for `OsTask`, `OsIsr`, `OsResource`, `OsAlarm`, and `OsEvent` definitions. The analysis uses these definitions and the supported multitasking keywords to configure tasks, interrupts, cyclical tasks, and critical sections.

To see how Polyspace models the `OsTask`, `OsIsr`, and `OsResource` definitions from your ARXML files, open the **Concurrency window** from the **Dashboard** pane. In that window, under the **Entry points** column, the names of the elements are extracted from their `<SHORT-NAME>` values in the ARXML files.

Off (default)

Polyspace does not set up a multitasking configuration for your AUTOSAR project.

Additional Considerations

- The analysis ignores `TerminateTask()` declarations in your source code and considers that subsequent code is executed.
- Polyspace supports multitasking configuration only from ARXML files for AUTOSAR specification version 4.0 and later.

Dependencies

To enable this option in the user interface of the desktop products, first select the option `External multitasking configuration`.

Command-Line Information

Parameter: `-autosar-multitasking`
Value: *`file1 [,file2, dir1,...]`*
Default: Off
Example (Bug Finder): `polyspace-bug-finder -sources` *`source_path`* `-I` *`include_path`* `-autosar-multitasking C:\Polyspace_Workspace\AUTOSAR\myFile.arxml`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`source_path`* `-I` *`include_path`* `-autosar-multitasking C:\Polyspace_Workspace\AUTOSAR\myFile.arxml`

See Also

`Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)` | `External multitasking configuration` | `OIL files selection (-osek-multitasking)`

Topics

“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”

Introduced in R2018a

Configure multitasking manually

Consider that code is intended for multitasking

Description

This option is not available for code generated from MATLAB code or Simulink models.

Specify whether your code is a multitasking application. This option allows you to manually configure the multitasking structure for Polyspace.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node.

Command line: See “Command-Line Information” on page 1-105.

Why Use This Option

By default, Bug Finder determines your multitasking model from your use of multithreading functions. In Code Prover, you have to enable automatic concurrency detection with the option `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`. However, in some cases, using automatic concurrency detection can slow down the Code Prover analysis.

In cases where automatic concurrency detection is not supported, you can explicitly specify your multitasking model by using this option. Once you select this option, you can explicitly specify your entry point functions, cyclic tasks, interrupts and protection mechanisms for shared variables, such as critical section details.

A Code Prover verification uses your specifications to determine:

- Whether a global variable is shared.

  See “Global Variables” (Polyspace Code Prover).
- Whether a run-time error can occur.

  For instance, if the operation `var++` occurs in the body of a cyclic task and you do not impose a limit on `var`, the operation can overflow. The analysis detects the possible overflow.

A Bug Finder analysis uses your specifications to look for concurrency defects. For more information, see “Concurrency Defects”.

Settings

☑ On

The code is intended for a multitasking application.

You have to explicitly specify your multitasking configuration using other Polyspace options. See “Configuring Polyspace Multitasking Analysis Manually”.

Off (default)

The code is not intended for a multitasking application.

Disabling the option has this additional effect in Code Prover:

- If a `main` exists, Code Prover verifies only those functions that are called by the `main`.
- If a `main` does not exist, Polyspace verifies the functions that you specify. To verify the functions, Polyspace generates a `main` function and calls functions from the generated `main` in a sequence that you specify. For more information, see `Verify module or library (-main-generator)`.

Tips

If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.

Command-Line Information

There is no single command-line option to turn on multitasking analysis. By using any of the options `Tasks (-entry-points)`, `Cyclic tasks (-cyclic-tasks)` or `Interrupts (-interrupts)`, you turn on multitasking analysis.

See Also

`-non-preemptable-tasks` | `-preemptable-interrupts` | `Critical section details (-critical-section-begin -critical-section-end)` | `Cyclic tasks (-cyclic-tasks)` | `Tasks (-entry-points)` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics

“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”
“Configuring Polyspace Multitasking Analysis Manually”
“Protections for Shared Variables in Multitasking Code”

Enable automatic concurrency detection for Code Prover (`-enable-concurrency-detection`)

Automatically detect certain families of multithreading functions

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify whether the analysis must automatically detect POSIX®, VxWorks®, Windows, µC/OS II and other multithreading functions.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See "Dependencies" (Polyspace Code Prover) for other options that you must enable or disable.

Command line: Use the option `-enable-concurrency-detection`. See "Command-Line Information" on page 1-107.

Why Use This Option

If you use this option, Polyspace determines your multitasking model from your use of multithreading functions. In Bug Finder, automatic concurrency detection is enabled by default. In Code Prover, you have to explicitly enable automatic concurrency detection.

In some cases, using automatic concurrency detection can slow down the Code Prover analysis. In those cases, you can choose to not enable this option and explicitly specify your multitasking model. See "Configuring Polyspace Multitasking Analysis Manually".

Settings

☑ On

If you use one of the supported functions for multitasking, the analysis automatically detects your multitasking model from your code.

For a list of supported multitasking functions and limitations in auto-detection of threads, see "Auto-Detection of Thread Creation and Critical Section in Polyspace".

☐ Off (default)

The analysis does not attempt to detect the multitasking model from your code.

If you want to manually configure your multitasking model, see "Configuring Polyspace Multitasking Analysis Manually".

Dependencies

If you enable this option, your code must contain a `main` function. You cannot use the Code Prover options to generate a `main`.

Command-Line Information

Parameter: `-enable-concurrency-detection`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-enable-concurrency-detection`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-enable-concurrency-detection`

See Also

Topics

“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”
“Auto-Detection of Thread Creation and Critical Section in Polyspace”

Tasks (-entry-points)

Specify functions that serve as tasks to your multitasking application

Description

This option is not available for code generated from MATLAB code or Simulink models.

Specify functions that serve as tasks to your code. If the function does not exist, the verification warns you and continues the verification.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 1-109 for other options you must also enable.

Command line: Use the option `-entry-points`. See “Command-Line Information” on page 1-109.

Why Use This Option

Use this option when your code is intended for multitasking.

To specify cyclic tasks and interrupts, use the options `Cyclic tasks (-cyclic-tasks)` and `Interrupts (-interrupts)`. Use this option to specify other tasks.

A Code Prover analysis uses your specifications to determine:

- Whether a global variable is shared.

  See “Global Variables” (Polyspace Code Prover).
- Whether a run-time error can occur.

  For instance, if the operation `var++` occurs in the body of a cyclic task and you do not impose a limit on `var`, the operation can overflow. The analysis detects the possible overflow.

A Bug Finder analysis uses your specifications to look for concurrency defects. For more information, see “Concurrency Defects”.

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

To enable this option in the user interface of the desktop products, first select the option `Configure multitasking manually`.

Tips

- In Code Prover, the functions representing entry points must have the form

  ```
  void functionName (void)
  ```

- If a function `func` takes arguments, you cannot use it directly as task. To use `func` as task:

  1. Create a new function `newFunc`. The declaration must be of the form `void newFunc (void)`.
  2. Declare arguments to `func` as `volatile` variables local to `newFunc`. Call `func` inside `newFunc`.
  3. Specify `newFunc` as a task.

- If you specify a function as a task, you must provide its definition. Otherwise, a Code Prover verification stops with the error message:

  ```
  task func_name must be a userdef function without parameters
  ```

  A Bug Finder analysis continues but does not consider the function as an entry point.

- If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.
- The Polyspace multitasking analysis assumes that a task cannot interrupt itself.

Command-Line Information

Parameter: `-entry-points`
No Default
Value: *`function1`*`[,`*`function2`*`[,...]]`
Example (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-entry-points func_1,func_2`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-entry-points func_1,func_2`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-entry-points func_1,func_2`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-entry-points func_1,func_2`

See Also

`-non-preemptable-tasks` | `-preemptable-interrupts` | `Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)`

Topics
“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”
“Configuring Polyspace Multitasking Analysis Manually”
“Protections for Shared Variables in Multitasking Code”

Cyclic tasks (`-cyclic-tasks`)

Specify functions that represent cyclic tasks

Description

This option is not available for code generated from MATLAB code or Simulink models.

Specify functions that represent cyclic tasks. The analysis assumes that operations in the function body:

- Can execute any number of times.
- Can be interrupted by noncyclic tasks, other cyclic tasks and interrupts. Noncyclic tasks are specified with the option `Tasks (-entry-points)` and interrupts are specified with the option `Interrupts (-interrupts)`.

  To model a cyclic task that cannot be interrupted by other cyclic tasks, specify the task as nonpreemptable. See `-non-preemptable-tasks`. For examples, see “Define Preemptable Interrupts and Nonpreemptable Tasks”.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 1-111 for other options you must also enable.

Command line: Use the option `-cyclic-tasks`. See “Command-Line Information” on page 1-112.

Why Use This Option

Use this option to specify cyclic tasks in your multitasking code. The functions that you specify must have the prototype:

```
void function_name(void);
```

A Code Prover verification uses your specifications to determine:

- Whether a global variable is shared.

  See “Global Variables” (Polyspace Code Prover).
- Whether a run-time error can occur.

  For instance, if the operation `var++` occurs in the body of a cyclic task and you do not impose a limit on `var`, the operation can overflow. The analysis detects the possible overflow.

A Bug Finder analysis uses your specifications to look for concurrency defects. For the `Data race` defect, the software establishes the following relations between preemptable tasks and other tasks.

- *Data race between two preemptable tasks*:

  Unless protected, two operations in different preemptable tasks can interfere with each other. If the operations use the same shared variable without protection, a data race can occur.

If both operations are atomic, to see the defect, you have to enable the checker **Data race including atomic operations**.

- *Data race between a preemptable task and a nonpreemptable task or interrupt*:
  - An atomic operation in a preemptable task cannot interfere with an operation in a nonpreemptable task or an interrupt. Even if the operations use the same shared variable without protection, a data race cannot occur.
  - A nonatomic operation in a preemptable task also cannot interfere with an operation in a nonpreemptable task or an interrupt. However, the latter operation can interrupt the former. Therefore, if the operations use the same shared variable without protection, a data race can occur.

For more information, see “Concurrency Defects”.

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

To enable this option in the user interface of the desktop products, first select the option `Configure multitasking manually`.

Tips

- In Code Prover, the functions representing cyclic tasks must have the form

  ```
  void functionName (void)
  ```

- If a function `func` takes arguments, you cannot use it directly as a cyclic task. To use `func` as cyclic task:

  1. Create a new function `newFunc`. The declaration must be of the form `void newFunc (void)`.
  2. Declare arguments to `func` as `volatile` variables local to `newFunc`. Call `func` inside `newFunc`.
  3. Specify `newFunc` as cyclic task.

- If you specify a function as a cyclic task, you must provide its definition. Otherwise, a Code Prover verification stops with the error message:

  ```
  task func_name must be a userdef function without parameters
  ```

  A Bug Finder analysis continues but does not consider the function as a cyclic task.

- If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.

- The Polyspace multitasking analysis assumes that a task cannot interrupt itself.

Command-Line Information

Parameter: `-cyclic-tasks`
No Default
Value: *`function1`*`[,`*`function2`*`[,...]]`
Example (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-cyclic-tasks func_1,func_2`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-cyclic-tasks func_1,func_2`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-cyclic-tasks func_1,func_2`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-cyclic-tasks func_1,func_2`

See Also

`-non-preemptable-tasks` | `-preemptable-interrupts` | `Interrupts (-interrupts)` | `Tasks (-entry-points)`

Topics

“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”
“Configuring Polyspace Multitasking Analysis Manually”
“Protections for Shared Variables in Multitasking Code”
“Define Preemptable Interrupts and Nonpreemptable Tasks”

Introduced in R2016b

Interrupts (-interrupts)

Specify functions that represent nonpreemptable interrupts

Description

This option is not available for code generated from MATLAB code or Simulink models.

Specify functions that represent nonpreemptable interrupts. The analysis assumes that operations in the function body:

- Can execute any number of times.
- Cannot be interrupted by noncyclic tasks, cyclic tasks or other interrupts. Noncyclic tasks are specified with the option `Tasks (-entry-points)` and cyclic tasks are specified with the option `Cyclic tasks (-cyclic-tasks)`.

  To model an interrupt that can be interrupted by other interrupts, specify the interrupt as preemptable. See `-preemptable-interrupts`. For examples, see "Define Preemptable Interrupts and Nonpreemptable Tasks".

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See "Dependencies" on page 1-114 for other options you must also enable.

Command line: Use the option `-interrupts`. See "Command-Line Information" on page 1-115.

Why Use This Option

Use this option to specify interrupts in your multitasking code. The functions that you specify must have the prototype:

```
void function_name(void);
```

A Code Prover verification uses your specifications to determine:

- Whether a global variable is shared.

  See "Global Variables" (Polyspace Code Prover).
- Whether a run-time error can occur.

  For instance, if the operation `var=INT_MAX;` occurs in an interrupt and `var++` occurs in the body of a task, an overflow can occur if the interrupt excepts before the operation in the task. The analysis detects the possible overflow.

A Bug Finder analysis uses your specifications to look for concurrency defects. For the `Data race` defect, the analysis establishes the following relations between interrupts and other tasks:

- *Data race between two interrupts*:

  Two operations in different interrupts cannot interfere with each other (unless one of the interrupts is preemptable). Even if the operations use the same shared variable without protection, a data race cannot occur.

- *Data race between an interrupt and another task*:
  - An operation in an interrupt cannot interfere with an atomic operation in any other task. Even if the operations use the same shared variable without protection, a data race cannot occur.
  - An operation in an interrupt can interfere with a nonatomic operation in any other task unless the other task is also a nonpreemptable interrupt. Therefore, if the operations use the same shared variable without protection, a data race can occur.

See “Concurrency Defects”.

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

To enable this option in the user interface of the desktop products, first select the option `Configure multitasking manually`.

Tips

- In Code Prover, the functions representing interrupts must have the form

  ```
  void functionName (void)
  ```

- If a function `func` takes arguments, you cannot use it directly as an interrupt. To use `func` as interrupt:

  1. Create a new function `newFunc`. The declaration must be of the form `void newFunc (void)`.
  2. Declare arguments to `func` as `volatile` variables local to `newFunc`. Call `func` inside `newFunc`.
  3. Specify `newFunc` as interrupt.

- If you specify a function as an interrupt, you must provide its definition. Otherwise, a Code Prover verification stops with the error message:

  ```
  task func_name must be a userdef function without parameters
  ```

  A Bug Finder analysis continues but does not consider the function as an interrupt.

- If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.
- The Polyspace multitasking analysis assumes that an interrupt cannot interrupt itself.

Command-Line Information

Parameter: -interrupts
No Default
Value: *function1*[,*function2*[,...]]
Example (Bug Finder): polyspace-bug-finder -sources *file_name* -interrupts func_1,func_2
Example (Code Prover): polyspace-code-prover -sources *file_name* -interrupts func_1,func_2
Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -interrupts func_1,func_2
Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -interrupts func_1,func_2

See Also

-non-preemptable-tasks | -preemptable-interrupts | Cyclic tasks (-cyclic-tasks) | Tasks (-entry-points)

Topics

“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”
“Configuring Polyspace Multitasking Analysis Manually”
“Protections for Shared Variables in Multitasking Code”
“Define Preemptable Interrupts and Nonpreemptable Tasks”

Introduced in R2016b

Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)

Specify routines that disable and reenable interrupts.

Description

This option affects a Bug Finder analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify a routine that disables all interrupts and a routine that reenables all interrupts.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 1-117 for other options you must also enable.

Command line: Use the option `-routine-disable-interrupts` and `-routine-enable-interrupts`. See “Command-Line Information” on page 1-118.

Why Use This Option

The analysis uses the information to look for data race defects. For instance, in the following code, the function `disable_all_interrupts` disables all interrupts until the function `enable_all_interrupts` is called. Even if `task`, `isr1` and `isr2` run concurrently, the operations `x=0` or `x=1` cannot interrupt the operation `x++`. There are no data race defects.

```
int x;

void isr1() {
   x = 0;
}

void isr2() {
   x = 1;
}

void task() {
   disable_all_interrupts();
   x++;
   enable_all_interrupts();
}
```

Settings

No Default

- In **Disabling routine**, enter the routine that disables all interrupts.
- In **Enabling routine**, enter the routine that reenables all interrupts.

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

To enable this option in the user interface of the desktop products, first select the option `Configure multitasking manually`.

Tips

- The routine that you specify for the option disables preemption by all:
  - Non-cyclic tasks.

    See `Tasks (-entry-points)`.
  - Cyclic tasks.

    See `Cyclic tasks (-cyclic-tasks)`.
  - Interrupts.

    See `Interrupts (-interrupts)`.

  In other words, the analysis considers that the body of operations between the disabling routine and the enabling routine is atomic and not interruptible at all.
- Protection via disabling interrupts is conceptually different from protection via critical sections.

  In the Polyspace multitasking model, to protect two sections of code *from each other* via critical sections, you have to embed them in the same critical section. In other words, you have to place the two sections between calls to the same lock and unlock function.

  For instance, suppose you use critical sections as follows:

  ```
  void isr1() {
     begin_critical_section();
     x = 0;
     end_critical_section();
  }

  void isr2() {
     x = 1;
  }

  void task() {
     begin_critical_section();
     x++;
     end_critical_section();
  }
  ```

  Here, the operation `x++` is protected from the operation `x=0` in `isr1`, but not from the operation `x=1` in `isr2`. If the function `begin_critical_section` disabled *all interrupts*, calling it before `x++` would have been sufficient to protect it.

Typically, you use one pair of routines in your code to disable and reenable interrupts, but you can have many pairs of lock and unlock functions that implement critical sections.

- The routines that disable and enable interrupts must be functions. For instance, if you define a function-like macro:

  ```
  #define disable_interrupt() interrupt_flag=0
  ```

  You cannot use the macro `disable_interrupt()` as routine disabling interrupts.

Command-Line Information

Parameter: `-routine-disable-interrupts | -routine-enable-interrupts`
No Default
Value: *`function_name`*
Example (Bug Finder): `polyspace-bug-finder -sources file_name -routine-disable-interrupts atomic_section_begins -routine-enable-interrupts atomic_section_ends`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -routine-disable-interrupts atomic_section_begins -routine-enable-interrupts atomic_section_ends`

See Also

`-non-preemptable-tasks | -preemptable-interrupts | Critical section details (-critical-section-begin -critical-section-end) | Cyclic tasks (-cyclic-tasks) | Interrupts (-interrupts) | Tasks (-entry-points) | Temporally exclusive tasks (-temporal-exclusions-file)`

Topics

“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”
“Configuring Polyspace Multitasking Analysis Manually”
“Protections for Shared Variables in Multitasking Code”
“Define Atomic Operations in Multitasking Code”
“Concurrency Defects”

Introduced in R2017a

Critical section details (-critical-section-begin -critical-section-end)

Specify functions that begin and end critical sections

Description

This option is not available for code generated from MATLAB code or Simulink models.

When verifying multitasking code, Polyspace considers that a critical section lies between calls to a lock function and an unlock function.

```
lock();
/* Critical section code */
unlock();
```

Specify the lock and unlock function names for your critical sections (for instance, `lock()` and `unlock()` in above example).

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 1-120 for other options you must also enable.

Command line: Use the option `-critical-section-begin` and `-critical-section-end`. See “Command-Line Information” on page 1-121.

Why Use This Option

When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait till `my_task` calls the corresponding unlock function. Therefore, critical section operations in the other tasks cannot interrupt critical section operations in `my_task`.

For instance, the operation `var++` in `my_task1` and `my_task2` cannot interrupt each other.

```
int var;

void my_task1() {
   my_lock();
   var++;
   my_unlock();
}

void my_task2() {
   my_lock();
   var++;
   my_unlock();
}
```

Using your specifications, a Code Prover verification checks if your placement of lock and unlock functions protects all shared variables from concurrent access. When determining values of those variables, the verification accounts for the fact that critical sections in different tasks do not interrupt each other.

A Bug Finder analysis uses the critical section information to look for concurrency defects such as data race and deadlock.

Settings

No Default

Click  to add a field.

- In **Starting routine**, enter name of lock function.
- In **Ending routine**, enter name of unlock function.

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

To enable this option in the user interface of the desktop products, first select the option `Configure multitasking manually`.

Tips

- You can also use primitives such as the POSIX functions `pthread_mutex_lock` and `pthread_mutex_unlock` to begin and end critical sections. For a list of primitives that Polyspace can detect automatically, see “Auto-Detection of Thread Creation and Critical Section in Polyspace”.
- For function calls that begin and end critical sections, Polyspace ignores the function arguments.

  For instance, Polyspace treats the two code sections below as the same critical section.

Starting routine: `my_lock`	
Ending routine: `my_unlock`	

`void my_task1() {`
`  my_lock(1);`
`  /* Critical section code */`
`  my_unlock(1);`
`}`
	`void my_task2() {`
	`  my_lock(2);`
	`  /* Critical section code */`
	`  my_unlock(2);`
	`}`

  To work around the limitation, see “Define Critical Sections with Functions That Take Arguments”.
- The functions that begin and end critical sections must be functions. For instance, if you define a function-like macro:

  ```
  #define init() num_locks++
  ```

  You cannot use the macro `init()` to begin or end a critical section.
- When you use multiple critical sections, you can run into issues such as:

- Deadlock: A sequence of calls to lock functions causes two tasks to block each other.
- Double lock: A lock function is called twice in a task without an intermediate call to an unlock function.

Use Polyspace Bug Finder to detect such issues. See “Concurrency Defects”.

Then, use Polyspace Code Prover™ to detect if your placement of lock and unlock functions actually protects all shared variables from concurrent access. See “Global Variables” (Polyspace Code Prover).

- When considering possible values of shared variables, a Code Prover verification takes into account your specifications for critical sections.

  However, if the shared variable is a pointer or array, the software uses the specifications only to determine if the variable is a shared protected global variable. For run-time error checking, the software does not take your specifications into account and considers that the variable can be concurrently accessed.

Command-Line Information

Parameter: `-critical-section-begin | -critical-section-end`
No Default
Value: *`function1`*`:cs1[,`*`function2`*`:cs2[,...]]`
Example (Bug Finder): `polyspace-bug_finder -sources` *`file_name`* `-critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`
Example (Bug Finder Server): `polyspace-bug_finder-server -sources` *`file_name`* `-critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`

See Also

`-non-preemptable-tasks | -preemptable-interrupts | Cyclic tasks (-cyclic-tasks) | Interrupts (-interrupts) | Tasks (-entry-points) | Temporally exclusive tasks (-temporal-exclusions-file)`

Topics

“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”
“Configuring Polyspace Multitasking Analysis Manually”
“Protections for Shared Variables in Multitasking Code”
“Define Atomic Operations in Multitasking Code”
“Define Critical Sections with Functions That Take Arguments”
“Concurrency Defects”
“Global Variables” (Polyspace Code Prover)

Temporally exclusive tasks (`-temporal-exclusions-file`)

Specify entry point functions that cannot execute concurrently

Description

This option is not available for code generated from MATLAB code or Simulink models.

Specify entry point functions that cannot execute concurrently. The execution of the functions cannot overlap with each other.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 1-122 for other options you must also enable.

Command line: Use the option `-temporal-exclusions-file`. See “Command-Line Information” on page 1-123.

Why Use This Option

Use this option to implement temporal exclusion in multitasking code.

A Code Prover verification checks if specifying certain tasks as temporally exclusive protects all shared variables from concurrent access. When determining possible values of those shared variables, the verification accounts for the fact that temporally exclusive tasks do not interrupt each other. See “Global Variables” (Polyspace Code Prover).

A Bug Finder analysis uses the temporal exclusion information to look for concurrency defects such as data race. See “Concurrency Defects”.

Settings

No Default

Click  to add a field. In each field, enter a space-separated list of functions. Polyspace considers that the functions in the list cannot execute concurrently.

Enter the function names manually or choose from a list.

- Click  to add a field and enter the function names.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

To enable this option in the user interface of the desktop products, first select the option `Configure multitasking manually`.

Tips

When considering possible values of shared variables, a Code Prover verification takes into account your specifications for temporally exclusive tasks.

However, if the shared variable is a pointer or array, the software uses the specifications only to determine if the variable is a shared protected global variable. For run-time error checking in Code Prover, the software does not take your specifications into account and considers that the variable can be concurrently accessed.

Command-Line Information

For the command-line option, create a temporal exclusions file in the following format:

- On each line, enter one group of temporally excluded tasks.
- Within a line, the tasks are separated by spaces.

To enter comments, begin with #. For an example, see the file *polyspaceroot*\polyspace\examples\cxx\Code_Prover_Example\sources\temporal_exclusions.txt. Here, *polyspaceroot* is the Polyspace installation folder, for example C:\Program Files\Polyspace\R2019a.
Parameter: -temporal-exclusions-file
No Default
Value: Name of temporal exclusions file
Example (Bug Finder): polyspace-bug-finder -sources *file_name* -temporal-exclusions-file "C:\exclusions_file.txt"
Example (Code Prover): polyspace-code-prover -sources *file_name* -temporal-exclusions-file "C:\exclusions_file.txt"
Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -temporal-exclusions-file "C:\exclusions_file.txt"
Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -temporal-exclusions-file "C:\exclusions_file.txt"

See Also

-non-preemptable-tasks | -preemptable-interrupts | Critical section details (-critical-section-begin -critical-section-end) | Cyclic tasks (-cyclic-tasks) | Interrupts (-interrupts) | Tasks (-entry-points)

Topics
“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”
“Configuring Polyspace Multitasking Analysis Manually”
“Protections for Shared Variables in Multitasking Code”
“Define Atomic Operations in Multitasking Code”
“Concurrency Defects”
“Global Variables” (Polyspace Code Prover)

Set checkers by file (`-checkers-selection-file`)

Define a custom set of coding standards checks for your analysis

Description

Specify the full path of a configuration XML file where you define custom selections of coding standards checkers. You can, in the same file, define a custom selection of checkers for each of these coding standards:

- MISRA C: 2004
- MISRA C: 2012
- MISRA C++
- JSF AV C++
- AUTOSAR C++14 *(Bug Finder only)*
- CERT® C *(Bug Finder only)*
- CERT C++ *(Bug Finder only)*
- ISO/IEC TS 17961 *(Bug Finder only)*

You can also define custom rules to match identifiers in your code against text patterns you specify.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node.

Command line: Use the option `-checkers-selection-file`. See “Command-Line Information” on page 1-126.

When you enable this option, set the coding standards you select to `from-file` to use the specified configuration file.

Why Use This Option

Use this option to define a selection of coding standard checkers specific to your organization. The configuration of different coding standards is consolidated in a single XML file which you can reuse across projects to enforce common coding standards.

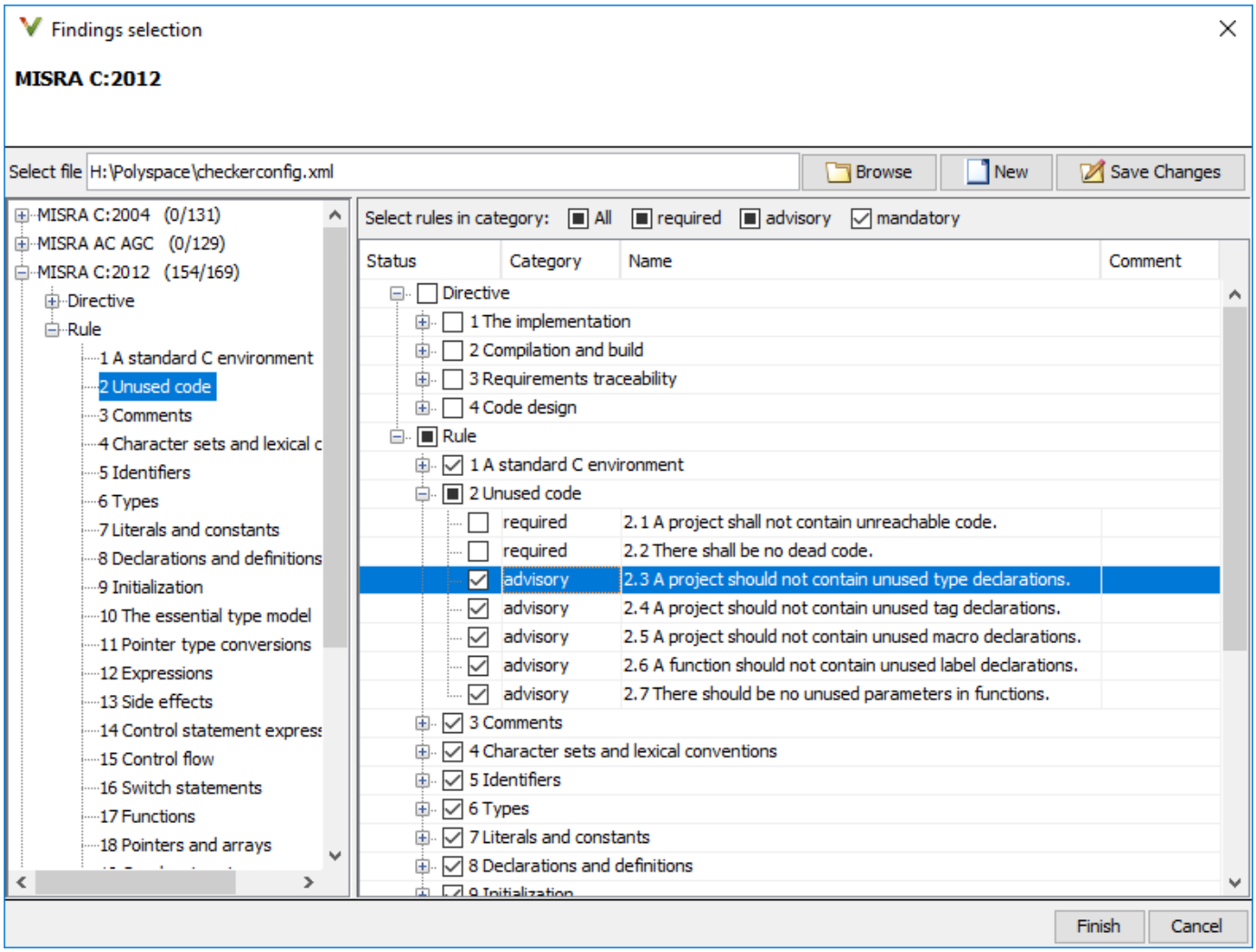
Settings

☑ On

Polyspace checks your code against the selection of coding standard checkers, or the custom rules, defined in the configuration file you specify.

To create a configuration file, open the **Findings selection** window by clicking  . In the left pane, choose the coding standard you want to configure, then select the rules you want to check for this coding standard in the right pane.

To use or update an existing file, enter the full path to the file in the field provided or click **Browse** in the **Findings selection** window.



Off (default)

Polyspace does not check your code against the selection of coding standard checkers, or the custom rules, defined in the configuration file you specify.

Tips

- With the Polyspace desktop products, specify the coding standard configuration in the user interface of the desktop products. When you save the configuration, an XML file is automatically created for use in the current and other projects.
- With the Polyspace Server products, you have to create a coding standard XML from scratch. Depending on the standard that you want to enable, make a writeable copy of one of the files in

polyspaceserverroot\help\toolbox\polyspace_bug_finder_server\examples\coding_standards_XML and turn off rules using entries in the XML file (all rules from a standard are enabled in the template). Here, *polyspaceserverroot* is the root installation folder for the Polyspace Server products, for instance, C:\Program Files\Polyspace Server\R2019a.

For instance, to turn off MISRA C: 2012 rule 8.1, use this entry in the file misra_c_2012_rules.xml:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="off">
      </check>
      ...
  </section>
  ...
</standard>
```

For full list of rule id-s and section names, see:

- "AUTOSAR C++14 Rules"
- "CERT C Rules and Recommendations"
- "CERT C++ Rules"
- "ISO/IEC TS 17961 Rules"
- "Custom Coding Rules"
- "JSF C++ Rules"
- "MISRA C:2004 Rules"
- "MISRA C:2012 Directives and Rules"
- "MISRA C++:2008 Rules"

Note The XML format of the checker configuration file can change in future releases.

Command-Line Information

Parameter: -checkers-selection-file
Value: Full path of XML configuration file
Default: Off
Example (Bug Finder): polyspace-bug-finder -sources *file_name* -checkers-selection-file "C:\Standards\custom_config.xml" -misra3 from-file
Example (Code Prover): polyspace-code-prover -sources *file_name* -checkers-selection-file "C:\Standards\custom_config.xml" -misra3 from-file
Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -checkers-selection-file "C:\Standards\custom_config.xml" -misra3 from-file
Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -checkers-selection-file "C:\Standards\custom_config.xml" -misra3 from-file

See Also

Do not generate results for (-do-not-generate-results-for)

Topics
“Specify Polyspace Analysis Options”
“Check for Coding Standard Violations”

Check MISRA C:2004 (`-misra2`)

Check for violation of MISRA C:2004 rules

Note Polyspace will no longer support custom configuration files in text format in a future release. See “Compatibility Considerations”.

Description

Specify whether to check for violation of MISRA C:2004 rules. Each value of the option corresponds to a subset of rules to check.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 1-129 for other options that you must also enable.

Command line: Use the option `-misra2`. See “Command-Line Information” on page 1-129.

Why Use This Option

Use this option to specify the subset of MISRA C:2004 rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

Settings

Default: `required-rules`

`required-rules`

Check required coding rules.

`single-unit-rules`

Check a subset of rules that apply only to single translation units. These rules are checked in the compilation phase of the analysis.

`system-decidable-rules`

Check rules in the `single-unit-rules` subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. These rules are checked in the compilation and linking phases of the analysis.

`all-rules`

Check required and advisory coding rules.

`SQO-subset1`

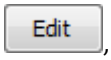
Check only a subset of MISRA C rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C:2004)”.

`SQO-subset2`

Check a subset of rules including `SQO-subset1` and some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C:2004)”.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click 

, then select the rules and recommendations you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

Dependencies

- This option is available only if you set `Source code language (-lang)` to `C` or `C-CPP`.

  For projects with mixed C and C++ code, the MISRA C:2004 checker analyzes only `.c` files.
- If you set `Source code language (-lang)` to `C-CPP`, you can activate a C coding rule checker **and** a C++ coding rule checker. When you have both C and C++ coding rule checkers active, to avoid duplicate results, Polyspace does not produce the C coding rules found in the linking phase (such as MISRA C:2012 Rule 8.3).

Tips

- To reduce unproven results in Polyspace Code Prover:

  1. Find coding rule violations in `SQO-subset1`. Fix your code to address the violations and rerun verification.
  2. Find coding rule violations in `SQO-subset2`. Fix your code to address the violations and rerun verification.
- If you select the option `single-unit-rules` or `system-decidable-rules` and choose to detect coding rule violations only, the analysis can complete quicker than checking other rules. For more information, see “Coding Rule Subsets Checked Early in Analysis”.

Command-Line Information

Parameter: `-misra2`
Value: `required-rules` | `all-rules` | `SQO-subset1` | `SQO-subset2` | `single-unit-rules` | `system-decidable-rules` | `from-file`
Default: `required-rules`
Example (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-misra2 all-rules`

Example (Code Prover): `polyspace-code-prover -sources file_name -misra2 all-rules`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -misra2 all-rules`
Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -misra2 all-rules`

Compatibility Considerations

Polyspace will no longer support text format for coding rules file
Not recommended starting in R2019a

Starting in R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your custom selection for each coding standard in separate text files. Polyspace will stop supporting custom coding standard files in text format in a future release.

Desktop interface:

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click  . In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *filename*`.xml`, where *filename* is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

Command-line:

If you do not have access to a Polyspace desktop interface, use the file `misra_c_2004_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in *polyspaceroot*`\help\toolbox\polyspace_bug_finder_server\examples\coding_standards_XML` . Here, *polyspaceroot* is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace\R2019a`. To update your script, see this table

Option	Use Instead
`-misra2 "custom_standard.conf"`	`-checkers-selection-file misra_c_2004_rules.xml -misra2 from-file`

.

Note The XML format of the checker configuration file can change in future releases.

Example of Configuration File in XML Format

To turn on MISRA C: 2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="on">
      </check>
      ...
  </section>
  ...
</standard>
```

For full list of rule id-s and section names, see:

- "AUTOSAR C++14 Rules"
- "CERT C Rules and Recommendations"
- "ISO/IEC TS 17961 Rules"
- "Custom Coding Rules"
- "JSF C++ Rules"
- "MISRA C:2004 Rules"
- "MISRA C:2012 Directives and Rules"
- "MISRA C++:2008 Rules"

See Also

`Do not generate results for (-do-not-generate-results-for)`

Topics
"Specify Polyspace Analysis Options"
"Check for Coding Standard Violations"
"MISRA C:2004 Rules"

Check MISRA AC AGC (`-misra-ac-agc`)

Check for violation of MISRA AC AGC rules

Note Polyspace will no longer support custom configuration files in text format in a future release. See “Compatibility Considerations”.

Description

Specify whether to check for violation of rules specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*. Each value of the option corresponds to a subset of rules to check.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 1-133 for other options that you must also enable.

Command line: Use the option `-misra-ac-agc`. See “Command-Line Information” on page 1-133.

Why Use This Option

Use this option to specify the subset of MISRA C:2004 AC AGC rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

Settings

Default: `OBL-rules`

`OBL-rules`

Check required coding rules.

`OBL-REC-rules`

Check required and recommended rules.

`single-unit-rules`

Check a subset of rules that apply only to single translation units. These rules are checked in the compilation phase of the analysis.

`system-decidable-rules`

Check rules in the `single-unit-rules` subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. These rules are checked in the compilation and linking phases of the analysis.

`all-rules`

Check required, recommended and readability-related rules.

`SQO-subset1`

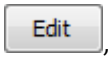
Check a subset of rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (AC AGC)”.

`SQO-subset2`

Check a subset of rules including `SQO-subset1` and some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (AC AGC)”.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click 

, then select the rules and recommendations you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

Dependencies

- This option is available only if you set `Source code language (-lang)` to `C` or `C-CPP`.

  For projects with mixed C and C++ code, the MISRA AC AGC checker analyzes only `.c` files.
- If you set `Source code language (-lang)` to `C-CPP`, you can activate a C coding rule checker **and** a C++ coding rule checker. When you have both C and C++ coding rule checkers active, to avoid duplicate results, Polyspace does not produce the C coding rules found in the linking phase (such as MISRA C:2012 Rule 8.3).

Tips

- To reduce unproven results in Polyspace Code Prover:

  1. Find coding rule violations in `SQO-subset1`. Fix your code to address the violations and rerun verification.
  2. Find coding rule violations in `SQO-subset2`. Fix your code to address the violations and rerun verification.
- If you select the option `single-unit-rules` or `system-decidable-rules` and choose to detect coding rule violations only, the analysis can complete quicker than checking other rules. For more information, see “Coding Rule Subsets Checked Early in Analysis”.

Command-Line Information

Parameter: `-misra-ac-agc`
Value: `OBL-rules` | `OBL-REC-rules` | `single-unit-rules` | `system-decidable-rules` | `all-rules` | `SQO-subset1` | `SQO-subset2` | `from-file`
Default: `OBL-rules`
Example (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-misra-ac-agc all-rules`

Example (Code Prover): `polyspace-code-prover -sources file_name -misra-ac-agc all-rules`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -misra-ac-agc all-rules`
Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -misra-ac-agc all-rules`

Compatibility Considerations

Polyspace will no longer support text format for coding rules file

Not recommended starting in R2019a

Starting in R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your custom selection for each coding standard in separate text files. Polyspace will stop supporting custom coding standard files in text format in a future release.

Desktop interface:

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click  . In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *filename*`.xml`, where *filename* is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

Command-line:

If you do not have access to a Polyspace desktop interface, use the file `misra_ac_agc_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in *polyspaceroot*`\help\toolbox\polyspace_bug_finder_server\examples\coding_standards_XML`. Here, *polyspaceroot* is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace\R2019a`. To update your script, see this table

Option	Use Instead
`-misra-ac-agc "custom_standard.conf"`	`-checkers-selection-file misra_ac_agc_rules.xml -misra-ac-agc from-file`

.

Note The XML format of the checker configuration file can change in future releases.

Example of Configuration File in XML Format

To turn on MISRA C: 2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="on">
      </check>
      ...
  </section>
  ...
</standard>
```

For full list of rule id-s and section names, see:

- "AUTOSAR C++14 Rules"
- "CERT C Rules and Recommendations"
- "ISO/IEC TS 17961 Rules"
- "Custom Coding Rules"
- "JSF C++ Rules"
- "MISRA C:2004 Rules"
- "MISRA C:2012 Directives and Rules"
- "MISRA C++:2008 Rules"

See Also

`Do not generate results for (-do-not-generate-results-for)`

Topics
"Specify Polyspace Analysis Options"
"Check for Coding Standard Violations"
"MISRA C:2004 Rules"

Check MISRA C:2012 (`-misra3`)

Check for violations of MISRA C:2012 rules and directives

Note Polyspace will no longer support custom configuration files in text format in a future release. See “Compatibility Considerations”.

Description

Specify whether to check for violations of MISRA C:2012 guidelines. Each value of the option corresponds to a subset of guidelines to check.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 1-137 for other options that you must also enable.

Command line: Use the option `-misra3`. See “Command-Line Information” on page 1-138.

Why Use This Option

Use this option to specify the subset of MISRA C:2012 rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

Settings

Default: `mandatory-required`

`mandatory`
: Check for mandatory guidelines.

`mandatory-required`
: Check for mandatory and required guidelines.

- Mandatory guidelines: Your code must comply with these guidelines.
- Required guidelines: You may deviate from these guidelines. However, you must complete a formal deviation record, and your deviation must be authorized.

  See Section 5.4 of the MISRA C:2012 guidelines. For an example of a deviation record, see Appendix I of the MISRA C:2012 guidelines.

Note To turn off some required guidelines, instead of `mandatory-required` select `custom`. To clear specific guidelines, click Edit. In the **Comment** column, enter your rationale for disabling a guideline. For instance, you can enter the Deviation ID that refers to a deviation record for the guideline. The rationale appears in your generated report.

`single-unit-rules`

Check a subset of rules that apply only to single translation units. These rules are checked in the compilation phase of the analysis.

`system-decidable-rules`

Check rules in the `single-unit-rules` subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. These rules are checked in the compilation and linking phases of the analysis.

`all`

Check for mandatory, required, and advisory guidelines.

`SQO-subset1`

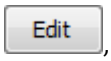
Check for only a subset of guidelines. In Polyspace Code Prover, observing these rules can reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C:2012)”.

`SQO-subset2`

Check for the subset `SQO-subset1`, plus some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C:2012)”.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click 

, then select the rules and recommendations you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

Dependencies

- This option is available only if you set `Source code language (-lang)` to `C` or `C-CPP`.

  For projects with mixed C and C++ code, the MISRA C:2012 checker analyzes only `.c` files.
- If you set `Source code language (-lang)` to `C-CPP`, you can activate a C coding rule checker **and** a C++ coding rule checker. When you have both C and C++ coding rule checkers active, to avoid duplicate results, Polyspace does not produce the C coding rules found in the linking phase (such as MISRA C:2012 Rule 8.3).

Tips

- To reduce unproven results in Polyspace Code Prover:

  1. Find coding rule violations in `SQO-subset1`. Fix your code to address the violations and rerun verification.
  2. Find coding rule violations in `SQO-subset2`. Fix your code to address the violations and rerun verification.

- If you select the option `single-unit-rules` or `system-decidable-rules` and choose to detect coding rule violations only, the analysis can complete quicker than checking other rules. For more information, see “Coding Rule Subsets Checked Early in Analysis”.
- Polyspace Code Prover does not support checking of the following:
  - MISRA C:2012 Directive 4.13 and 4.14
  - MISRA C:2012 Rule 21.13, 21.14, and 21.17 - 21.20
  - MISRA C:2012 Rule 22.1 - 22.4 and 22.6 - 22.10

  For support of all MISRA C: 2012 rules including the security guidelines in Amendment 1, use Polyspace Bug Finder.

Command-Line Information

Parameter: `-misra3`
Value: `mandatory` | `mandatory-required` | `single-unit-rules` | `system-decidable-rules` | `all` | `SQO-subset1` | `SQO-subset2` | `from-file`
Default: `mandatory-required`
Example (Bug Finder): `polyspace-bug-finder -lang c -sources file_name -misra3 mandatory-required`
Example (Code Prover): `polyspace-code-prover -lang c -sources file_name -misra3 mandatory-required`
Example (Bug Finder Server): `polyspace-bug-finder-server -lang c -sources file_name -misra3 mandatory-required`
Example (Code Prover Server): `polyspace-code-prover-server -lang c -sources file_name -misra3 mandatory-required`

Compatibility Considerations

Polyspace will no longer support text format for coding rules file

Not recommended starting in R2019a

Starting in R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your custom selection for each coding standard in separate text files. Polyspace will stop supporting custom coding standard files in text format in a future release.

Desktop interface:

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click  . In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *filename*`.xml`, where *filename* is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

Command-line:

If you do not have access to a Polyspace desktop interface, use the file `misra_c_2012_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in *`polyspaceroot`*`\help\toolbox\polyspace_bug_finder_server\examples\coding_standards_XML`. Here, *`polyspaceroot`* is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace\R2019a`. To update your script, see this table

Option	Use Instead
`-misra3 "custom_standard.conf"`	`-checkers-selection-file misra_c_2012_rules.xml -misra3 from-file`

.

Note The XML format of the checker configuration file can change in future releases.

Example of Configuration File in XML Format

To turn on MISRA C: 2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="on">
      </check>
      ...
  </section>
  ...
</standard>
```

For full list of rule id-s and section names, see:

- “AUTOSAR C++14 Rules”
- “CERT C Rules and Recommendations”
- “ISO/IEC TS 17961 Rules”
- “Custom Coding Rules”
- “JSF C++ Rules”
- “MISRA C:2004 Rules”
- “MISRA C:2012 Directives and Rules”
- “MISRA C++:2008 Rules”

See Also

`Do not generate results for (-do-not-generate-results-for)`

Topics

“Specify Polyspace Analysis Options”

“Check for Coding Standard Violations”
“MISRA C:2012 Directives and Rules”

Use generated code requirements (-misra3-agc-mode)

Check for violations of MISRA C:2012 rules and directives that apply to generated code

Description

Specify whether to use the MISRA C:2012 categories for automatically generated code. This option changes which rules are mandatory, required, or advisory.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependency” on page 1-142 for other options that you must also enable.

Command line: Use the option `-misra3-agc-mode`. See “Command-Line Information” on page 1-142.

Why Use This Option

Use this option to specify that you are checking for MISRA C:2012 rules in generated code. The option modifies the MISRA C:2012 subsets so that they are tailored for generated code.

Settings

☐ Off (default)

Use the normal categories (mandatory, required, advisory) for MISRA C:2012 coding guideline checking.

☑ On (default for analyses from Simulink)

Use the generated code categories (mandatory, required, advisory, readability) for MISRA C:2012 coding guideline checking.

For analyses started from the Simulink plug-in, this option is the default value.

Category changed to `Advisory`

These rules are changed to advisory:

- 5.3
- 7.1
- 8.4, 8.5, 8.14
- 10.1, 10.2, 10.3, 10.4, 10.6, 10.7, 10.8
- 14.1, 14.4
- 15.2, 15.3
- 16.1, 16.2, 16.3, 16.4, 16.5, 16.6, 16.7

- 20.8

Category changed to `Readability`

These guidelines are changed to readability:

- Dir 4.5
- 2.3, 2.4, 2.5, 2.6, 2.7
- 5.9
- 7.2, 7.3
- 9.2, 9.3, 9.5
- 11.9
- 13.3
- 14.2
- 15.7
- 17.5, 17.7, 17.8
- 18.5
- 20.5

Dependency

To use this option, first select the `Check MISRA C:2012 (-misra3)` option.

Command-Line Information

Parameter: `-misra3-agc-mode`
Default: Off
Example (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-misra3 all -misra3-agc-mode`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-misra3 all -misra3-agc-mode`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-misra3 all -misra3-agc-mode`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-misra3 all -misra3-agc-mode`

See Also

`Check MISRA C:2012 (-misra3)` | `Do not generate results for (-do-not-generate-results-for)`

Topics

“Specify Polyspace Analysis Options”
“Check for Coding Standard Violations”
“MISRA C:2012 Directives and Rules”

Effective boolean types (-boolean-types)

Specify data types that coding rule checker must treat as effectively Boolean

Description

Specify data types that the coding rule checker must treat as effectively Boolean. You can specify a data type as effectively Boolean only if you have defined it through an `enum` or `typedef` statement in your source code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 1-144 for other options that you must also enable.

Command line: Use the option `-boolean-types`. See “Command-Line Information” on page 1-144.

Why Use This Option

Use this option to allow Polyspace to check the following coding rules:

- MISRA C: 2004 and MISRA AC AGC

Rule Number	Rule Statement
12.6	Operands of logical operators, &&, `||`, and `!`, should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to other operators.
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
15.4	A `switch` expression should not represent a value that is effectively Boolean.

- MISRA C: 2012

Rule Number	Rule Statement
10.1	Operands shall not be of an inappropriate essential type
10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category
10.5	The value of an expression should not be cast to an inappropriate essential type
14.4	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.
16.7	A switch-expression shall not have essentially Boolean type.

For example, in the following code, unless you specify `myBool` as effectively Boolean, Polyspace detects a violation of MISRA C: 2012 rule 14.4.

```
typedef int myBool;
```

```
void func1(void);
void func2(void);

void func(myBool flag) {
    if(flag)
        func1();
    else
        func2();
}
```

Settings

No Default

Click  to add a field. Enter a type name that you want Polyspace to treat as Boolean.

Dependencies

This option is enabled only if you select one of these options:

- `Check MISRA C:2004 (-misra2)`
- `Check MISRA AC AGC (-misra-ac-agc)`.
- `Check MISRA C:2012 (-misra3)`

Command-Line Information

Parameter: `-boolean-types`
Value: *`type1`*`[,`*`type2`*`[,...]]`
No Default
Example (Bug Finder): `polyspace-bug-finder -sources` *`filename`* `-misra2 required-rules -boolean-types boolean1_t,boolean2_t`
Example (Code Prover): `polyspace-code-prover -sources` *`filename`* `-misra2 required-rules -boolean-types boolean1_t,boolean2_t`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`filename`* `-misra2 required-rules -boolean-types boolean1_t,boolean2_t`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`filename`* `-misra2 required-rules -boolean-types boolean1_t,boolean2_t`

See Also

`Check MISRA AC AGC (-misra-ac-agc)` | `Check MISRA C:2004 (-misra2)` | `Check MISRA C:2012 (-misra3)`

Topics
“Specify Polyspace Analysis Options”
“Check for Coding Standard Violations”
“MISRA C:2004 Rules”
“MISRA C:2012 Directives and Rules”

Allowed pragmas (-allowed-pragmas)

Specify pragma directives that are documented

Description

Specify pragma directives that must not be flagged by MISRA C:2004 rule 3.4 or MISRA C++ rule 16-6-1. These rules require that you document all pragma directives.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 1-145 for other options that you must also enable.

Command line: Use the option `-allowed-pragmas`. See “Command-Line Information” on page 1-145.

Why Use This Option

MISRA C:2004/MISRA AC AGC rule 3.4 and MISRA C++ rule 16-6-1 require that all pragma directives are documented within the documentation of the compiler. If you list a pragma as documented using this analysis option, Polyspace does not flag use of the pragma as a violation of these rules.

Settings

No Default

Click  to add a field. Enter the pragma name that you want Polyspace to ignore during coding rule checking .

Dependencies

This option is enabled only if you select one of these options:

- `Check MISRA C:2004 (-misra2)`
- `Check MISRA AC AGC (-misra-ac-agc).`
- `Check MISRA C++:2008 (-misra-cpp)`

Command-Line Information

Parameter: `-allowed-pragmas`
Value: *`pragma1`*`[,`*`pragma2`*`[,...]]`
No Default
Example (Bug Finder): `polyspace-bug-finder -sources` *`filename`* `-misra-cpp required-rules -allowed-pragmas pragma_01,pragma_02`
Example (Code Prover): `polyspace-code-prover -sources` *`filename`* `-misra-cpp required-rules -allowed-pragmas pragma_01,pragma_02`

Example (Bug Finder Server): `polyspace-bug-finder-server -sources filename -misra-cpp required-rules -allowed-pragmas pragma_01,pragma_02`
Example (Code Prover Server): `polyspace-code-prover-server -sources filename -misra-cpp required-rules -allowed-pragmas pragma_01,pragma_02`

See Also

`Check MISRA AC AGC (-misra-ac-agc)` | `Check MISRA C++:2008 (-misra-cpp)` | `Check MISRA C:2004 (-misra2)`

Topics

“Specify Polyspace Analysis Options”
“Check for Coding Standard Violations”
“MISRA C:2004 Rules”
“MISRA C++:2008 Rules”

Check custom rules (-custom-rules)

Follow naming conventions for identifiers

Note Polyspace will no longer support custom configuration files in text format in a future release. See “Compatibility Considerations”.

Description

Define naming conventions for identifiers and check your code against them.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node.

Command line: Use the option `-custom-rules`. See “Command-Line Information” on page 1-149.

Why Use This Option

Use this option to impose naming conventions on identifiers. Using a naming convention allows you to easily determine the nature of an identifier from its name. For instance, if you define a naming convention for structures, you can easily tell whether an identifier represents a structured variable or not.

After analysis, the **Results List** pane lists violations of the naming conventions. On the **Source** pane, for every violation, Polyspace assigns a ▼ symbol to the keyword or identifier relevant to the violation.

Settings

☑ On

Polyspace matches identifiers in your code against text patterns you define. Define the text patterns in a custom coding rules file. To create a coding rules file,

- Use the custom rules wizard:

  1. Click Edit. A **Findings selection** window opens.
  2. The **Custom** node in the left pane is highlighted. Expand the nodes in the right pane to select custom rule you want to check.
  3. For every custom rule you want to check:

     a. Select **On** ☑.

     b. In the **Convention** column, enter the error message you want to display if the rule is violated.

     For example, for rule 4.3, **All struct fields must follow the specified pattern**, you can enter `All struct fields must begin with s_`. This message appears on the **Result Details** pane if:

- You specify the **Pattern** as `s_[A-Za-z0-9_]+`.
- A structure field in your code does not begin with `s_`.

c In the **Pattern** column, enter the text pattern.

For example, for rule 4.3, **All struct fields must follow the specified pattern**, you can enter `s_[A-Za-z0-9_]+`. Polyspace reports violation of rule 4.3 if a structure field does not begin with `s_`.

You can use Perl regular expressions to define patterns. For instance, you can use the following expressions.

Expression	Meaning
.	Matches any single character except newline
`[a-z0-9]`	Matches any single letter in the set `a-z`, or digit in the set `0-9`
`[^a-e]`	Matches any single letter not in the set `a-e`
`\d`	Matches any single digit
`\w`	Matches any single alphanumeric character or _
`x?`	Matches 0 or 1 occurrence of `x`
`x*`	Matches 0 or more occurrences of `x`
`x+`	Matches 1 or more occurrences of `x`

For frequent patterns, you can use the following regular expressions:

- `(?!__)[a-z0-9_]+(?!__)`, matches a text pattern that does not start and end with two underscores.

  ```
  int __text; //Does not match
  int _text_; //Matches
  ```

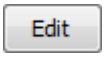
- `[a-z0-9_]+_(u8|u16|u32|s8|s16|s32)` , matches a text pattern that ends with a specific suffix.

  ```
  int _text_; //Does not match
  int _text_s16; //Matches
  int _text_s33; // Does not match
  ```

- `[a-z0-9_]+_(u8|u16|u32|s8|s16|s32)(_b3|_b8)?` , matches a text pattern that ends with a specific suffix and an optional second suffix.

  ```
  int _text_s16; //Matches
  int _text_s16_b8; //Matches
  ```

For a complete list of regular expressions, see Perl documentation.

To use or update an existing coding rules file, click 

to open the **Findings selection** window then do one of the following:

- Enter the full path to the file in the field provided
- Click **Browse** and navigate to the file location.

Off (default)

Polyspace does not check your code against custom naming conventions.

Command-Line Information

Parameter: `-custom-rules`
Value: `from-file`, specify the file using `Set checkers by file (-checkers-selection-file)`
Default: Off
Example (Bug Finder): `polyspace-bug-finder -sources file_name -custom-rules from-file -checkers-selection-file "C:\Standards\custom_config.xml"`
Example (Code Prover): `polyspace-code-prover -sources file_name -custom-rules from-file -checkers-selection-file "C:\Standards\custom_config.xml"`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -custom-rules from-file -checkers-selection-file "C:\Standards\custom_config.xml"`
Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -custom-rules from-file -checkers-selection-file "C:\Standards\custom_config.xml"`

Compatibility Considerations

Polyspace will no longer support text format for coding rules file

Not recommended starting in R2019a

Starting in R2019a, the file where you define custom coding rules uses the XML format. You can save selections for custom coding rules and all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your selection for each coding standard and custom coding rules in separate text files. Polyspace will stop supporting custom coding rule files in text format in a future release.

Desktop user interface:

If you have a project that contains custom coding rules and coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click  . In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *filename*`.xml`, where *filename* is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

Command-line:

If you do not have access to a Polyspace desktop interface, use the file `custom_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in *polyspaceroot*`\help\toolbox\polyspace_bug_finder_server\examples\coding_standards_XML`. Here, *polyspaceroot* is the root installation folder for the

Polyspace products, for instance, `C:\Program Files\Polyspace\R2019a`. To update your script, replace reference to the old file format with the new XML file format .

Example of Configuration File in XML Format

To turn on and define custom coding rule 8.1, use this entry:

```
<standard name="CUSTOM RULES">
  ...
  <section name="8 Constants">
      ...
      <check id="8.1" state="on">
       </check>
      ...
  </section>
  ...
</standard>
```

For full list of rule id-s and section names, see:

- “AUTOSAR C++14 Rules”
- “CERT C Rules and Recommendations”
- “ISO/IEC TS 17961 Rules”
- “Custom Coding Rules”
- “JSF C++ Rules”
- “MISRA C:2004 Rules”
- “MISRA C:2012 Directives and Rules”
- “MISRA C++:2008 Rules”

See Also

Topics

“Specify Polyspace Analysis Options”
“Check for Coding Standard Violations”
“Create Custom Coding Rules”
“Custom Coding Rules”

Check MISRA C++:2008 (-misra-cpp)

Check for violations of MISRA C++ rules

Note Polyspace will no longer support custom configuration files in text format in a future release. See “Compatibility Considerations”.

Description

Specify whether to check for violation of MISRA C++ rules. Each value of the option corresponds to a subset of rules to check.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependency” on page 1-152 for other options that you must also enable.

Command line: Use the option `-misra-cpp`. See “Command-Line Information” on page 1-152.

Why Use This Option

Use this option to specify the subset of MISRA C++ rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a symbol to the keyword or identifier relevant to the violation.

Settings

Default: `required-rules`

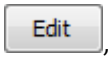
`required-rules`
: Check required coding rules.

`all-rules`
: Check required and advisory coding rules.

`SQO-subset1`
: Check only a subset of MISRA C++ rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C++)”.

`SQO-subset2`
: Check a subset of rules including `SQO-subset1` and some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (C++)”

`from-file`
: Specify an XML file where you configure a custom selection of checkers for this coding standard.

  To create a configuration file, click 

, then select the rules and recommendations you want

to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

Dependency

This option is available only if you set `Source code language (-lang)` to `CPP` or `C-CPP`.

For projects with mixed C and C++ code, the MISRA C++ checker analyzes only `.cpp` files.

Command-Line Information

Parameter: `-misra-cpp`
Value: `required-rules` | `all-rules` | `SQO-subset1` | `SQO-subset2` | `from-file`
Default: `required-rules`
Example (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-misra-cpp all-rules`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-misra-cpp all-rules`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-misra-cpp all-rules`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-misra-cpp all-rules`

Compatibility Considerations

Polyspace will no longer support text format for coding rules file

Not recommended starting in R2019a

Starting in R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your custom selection for each coding standard in separate text files. Polyspace will stop supporting custom coding standard files in text format in a future release.

Desktop interface:

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click  . In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *`filename`*`.xml`, where *`filename`* is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

Command-line:

If you do not have access to a Polyspace desktop interface, use the file `misra_cpp_2008_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in *`polyspaceroot`*`\help\toolbox\polyspace_bug_finder_server\examples\coding_standards_XML`. Here, *`polyspaceroot`* is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace\R2019a`. To update your script, see this table

Option	Use Instead
`-misra-cpp "custom_standard.conf"`	`-checkers-selection-file misra_cpp_2008_rules.xml -misra-cpp from-file`

.

Note The XML format of the checker configuration file can change in future releases.

Example of Configuration File in XML Format

To turn on MISRA C: 2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="on">
      </check>
      ...
  </section>
  ...
</standard>
```

For full list of rule id-s and section names, see:

- “AUTOSAR C++14 Rules”
- “CERT C Rules and Recommendations”
- “ISO/IEC TS 17961 Rules”
- “Custom Coding Rules”
- “JSF C++ Rules”
- “MISRA C:2004 Rules”
- “MISRA C:2012 Directives and Rules”
- “MISRA C++:2008 Rules”

See Also

`Do not generate results for (-do-not-generate-results-for)`

Topics

“Specify Polyspace Analysis Options”

“Check for Coding Standard Violations”
“MISRA C++:2008 Rules”

Check JSF AV C++ rules (-jsf-coding-rules)

Check for violations of JSF C++ rules

Note Polyspace will no longer support custom configuration files in text format in a future release. See “Compatibility Considerations”.

Description

Specify whether to check for violation of JSF AV C++ rules (JSF++:2005). Each value of the option corresponds to a subset of rules to check.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependency” on page 1-156 for other options that you must also enable.

Command line: Use the option `-jsf-coding-rules`. See “Command-Line Information” on page 1-156.

Why Use This Option

Use this option to specify the subset of JSF C++ rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

Settings

Default: `shall-rules`

`shall-rules`

Check all **Shall** rules. **Shall** rules are mandatory requirements and require verification.

`shall-will-rules`

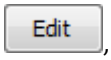
Check all **Shall** and **Will** rules. **Will** rules are intended to be mandatory requirements but do not require verification.

`all-rules`

Check all **Shall**, **Will**, and **Should** rules. **Should** rules are advisory rules.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click 

, then select the rules and recommendations you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

Tips

- If your project uses a setting other than `generic` for `Compiler (-compiler)`, some rules might not be completely checked. For example, AV Rule 8: "All code shall conform to ISO/IEC 14882:2002(E) standard C++."

Dependency

This option is available only if you set `Source code language (-lang)` to `CPP` or `C-CPP`.

For projects with mixed C and C++ code, the JSF C++ checker analyzes only `.cpp` files.

Command-Line Information

Parameter: `-jsf-coding-rules`
Value: `shall-rules` | `shall-will-rules` | `all-rules` | `from-file`
Default: `shall-rules`
Example (Bug Finder): `polyspace-bug-finder -sources file_name -jsf-coding-rules all-rules`
Example (Code Prover): `polyspace-code-prover -sources file_name -jsf-coding-rules all-rules`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -jsf-coding-rules all-rules`
Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -jsf-coding-rules all-rules`

Compatibility Considerations

Polyspace will no longer support text format for coding rules file

Not recommended starting in R2019a

Starting in R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your custom selection for each coding standard in separate text files. Polyspace will stop supporting custom coding standard files in text format in a future release.

Desktop interface:

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click  . In the **Findings selection** window, select the files then

click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *filename*.xml, where *filename* is the name of the first selected file alphabetically. For instance, if you select foo.conf and bar.conf, they are saved as bar.conf.xml.

Command-line:

If you do not have access to a Polyspace desktop interface, use the file StandardsConfiguration.xml as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in *polyspaceserverroot*\polyspace\examples\cxx\Bug_Finder_Example\sources or *polyspaceserverroot*\polyspace\examples\cxx\Code_Prover_Example\sources. Here, *polyspaceserverroot* is the root installation folder for the Polyspace products, for instance, C:\Program Files\Polyspace\R2019a. To update your script, see this table

Option	Use Instead
-jsf-coding-rules "custom_standard.conf"	-checkers-selection-file "custom_standard.conf.xml" -jsf-coding-rules from-file

.

Example of Configuration File in XML Format

To turn on MISRA C: 2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="on">
      </check>
      ...
  </section>
  ...
</standard>
```

For full list of rule id-s and section names, see:

- “AUTOSAR C++14 Rules”
- “CERT C Rules and Recommendations”
- “ISO/IEC TS 17961 Rules”
- “Custom Coding Rules”
- “JSF C++ Rules”
- “MISRA C:2004 Rules”
- “MISRA C:2012 Directives and Rules”
- “MISRA C++:2008 Rules”

See Also

Do not generate results for (-do-not-generate-results-for)

Topics

“Specify Polyspace Analysis Options”

“Check for Coding Standard Violations”
“JSF C++ Rules”

Check AUTOSAR C++ 14 (-autosar-cpp14)

Check for violations of AUTOSAR C++ 14 rules

Description

This option affects Bug Finder only.

Specify whether to check for violations of AUTOSAR C++ 14. Each value of the option corresponds to a subset of guidelines to check.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 1-160 for other options that you must also enable.

Command line: Use the option `-autosar-cpp14`. See “Command-Line Information” on page 1-160.

Why Use This Option

Use this option to specify the subset of AUTOSAR C++ 14 rules to check for[1].

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding standard violation, Polyspace assigns a  symbol to the keyword or identifier relevant to the violation.

Settings

Default: `all`

`all`

Check for violations of all AUTOSAR C++ 14 rules supported by Polyspace.

See “AUTOSAR C++14 Rules”.

`required`

Check for violations of *required* rules.

These rules are mandatory requirements placed on your code.

`automated`

Check for violations of *automated* rules.

You can automatically enforce these rules by means of static analysis.

1. The Polyspace checkers for AUTOSAR C++14 rules supports AUTOSAR C++14 release 18-03 (March 2018). Out of 390 rules from the standard, 247 rules are supported.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click Edit, then select the rules you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

Dependencies

- This option is available only if you set `Source code language (-lang)` to `CPP` or `C-CPP`.

Command-Line Information

Parameter: `-autosar-cpp14`
Value: `all` | `required` | `automated` | `from-file`
Default: `all`
Example (Bug Finder): `polyspace-bug-finder -lang cpp -sources` *`file_name`* `-autosar-cpp14 required`
Example (Bug Finder Server): `polyspace-bug-finder-server -lang cpp -sources` *`file_name`* `-autosar-cpp14 required`

See Also

`Do not generate results for (-do-not-generate-results-for)`

Topics

“Specify Polyspace Analysis Options”
“Check for Coding Standard Violations”
“AUTOSAR C++14 Rules”

Check SEI CERT-C (-cert-c)

Check for violations of CERT C rules and recommendations

Description

This option affects Bug Finder only.

Specify whether to check for violations of CERT C rules and recommendations. Each value of the option corresponds to a subset of the coding standard to check.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 1-168 for other options that you must also enable.

Command line: Use the option `-cert-c`. See “Command-Line Information” on page 1-168.

Why Use This Option

Use this option to specify the subset of CERT C rules and recommendations to check in your code.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding standard violation, Polyspace assigns a  symbol to the keyword or identifier relevant to the violation.

Settings

Default: `all`

`all-rules`

Check for violations of CERT C rules only.

See the CERT C website for an explanation of the difference between rules and recommendations.

List of CERT-C rules that Polyspace checks when you use `all-rules`

CERT C: Rule ARR30-C
CERT C: Rule ARR32-C
CERT C: Rule ARR36-C
CERT C: Rule ARR37-C
CERT C: Rule ARR38-C
CERT C: Rule ARR39-C
CERT C: Rule CON30-C
CERT C: Rule CON31-C
CERT C: Rule CON32-C

CERT C: Rule CON33-C
CERT C: Rule CON35-C
CERT C: Rule CON36-C
CERT C: Rule CON37-C
CERT C: Rule CON40-C
CERT C: Rule CON41-C
CERT C: Rule CON43-C
CERT C: Rule DCL30-C
CERT C: Rule DCL31-C
CERT C: Rule DCL36-C
CERT C: Rule DCL37-C
CERT C: Rule DCL38-C
CERT C: Rule DCL39-C
CERT C: Rule DCL40-C
CERT C: Rule DCL41-C
CERT C: Rule ENV30-C
CERT C: Rule ENV31-C
CERT C: Rule ENV32-C
CERT C: Rule ENV33-C
CERT C: Rule ENV34-C
CERT C: Rule ERR30-C
CERT C: Rule ERR32-C
CERT C: Rule ERR33-C
CERT C: Rule ERR34-C
CERT C: Rule EXP30-C
CERT C: Rule EXP32-C
CERT C: Rule EXP33-C
CERT C: Rule EXP34-C
CERT C: Rule EXP35-C
CERT C: Rule EXP36-C
CERT C: Rule EXP37-C
CERT C: Rule EXP39-C
CERT C: Rule EXP40-C
CERT C: Rule EXP42-C
CERT C: Rule EXP43-C
CERT C: Rule EXP44-C
CERT C: Rule EXP45-C

CERT C: Rule EXP46-C
CERT C: Rule EXP47-C
CERT C: Rule FIO30-C
CERT C: Rule FIO32-C
CERT C: Rule FIO34-C
CERT C: Rule FIO37-C
CERT C: Rule FIO38-C
CERT C: Rule FIO39-C
CERT C: Rule FIO40-C
CERT C: Rule FIO41-C
CERT C: Rule FIO42-C
CERT C: Rule FIO44-C
CERT C: Rule FIO45-C
CERT C: Rule FIO46-C
CERT C: Rule FIO47-C
CERT C: Rule FLP30-C
CERT C: Rule FLP32-C
CERT C: Rule FLP34-C
CERT C: Rule FLP36-C
CERT C: Rule FLP37-C
CERT C: Rule INT30-C
CERT C: Rule INT31-C
CERT C: Rule INT32-C
CERT C: Rule INT33-C
CERT C: Rule INT34-C
CERT C: Rule INT35-C
CERT C: Rule INT36-C
CERT C: Rule MEM30-C
CERT C: Rule MEM31-C
CERT C: Rule MEM33-C
CERT C: Rule MEM34-C
CERT C: Rule MEM35-C
CERT C: Rule MEM36-C
CERT C: Rule MSC30-C
CERT C: Rule MSC32-C
CERT C: Rule MSC33-C
CERT C: Rule MSC37-C

CERT C: Rule MSC38-C
CERT C: Rule MSC39-C
CERT C: Rule MSC40-C
CERT C: Rule POS30-C
CERT C: Rule POS33-C
CERT C: Rule POS34-C
CERT C: Rule POS35-C
CERT C: Rule POS36-C
CERT C: Rule POS37-C
CERT C: Rule POS38-C
CERT C: Rule POS39-C
CERT C: Rule POS44-C
CERT C: Rule POS48-C
CERT C: Rule POS49-C
CERT C: Rule POS51-C
CERT C: Rule POS52-C
CERT C: Rule POS54-C
CERT C: Rule PRE30-C
CERT C: Rule PRE31-C
CERT C: Rule PRE32-C
CERT C: Rule SIG30-C
CERT C: Rule SIG31-C
CERT C: Rule SIG34-C
CERT C: Rule SIG35-C
CERT C: Rule STR30-C
CERT C: Rule STR31-C
CERT C: Rule STR32-C
CERT C: Rule STR34-C
CERT C: Rule STR37-C
CERT C: Rule STR38-C
CERT C: Rule WIN30-C

`publish-2016`

Check for violations of CERT C rules only, as defined in the 2016 edition of the **SEI CERT C Coding Standard**.

See the CERT C website for an explanation of the difference between rules and recommendations.

List of CERT-C rules that Polyspace checks when you use `publish-2016`

CERT C: Rule ARR30-C
CERT C: Rule ARR32-C
CERT C: Rule ARR36-C
CERT C: Rule ARR37-C
CERT C: Rule ARR38-C
CERT C: Rule ARR39-C
CERT C: Rule CON30-C
CERT C: Rule CON31-C
CERT C: Rule CON32-C
CERT C: Rule CON33-C
CERT C: Rule CON35-C
CERT C: Rule CON36-C
CERT C: Rule CON37-C
CERT C: Rule CON40-C
CERT C: Rule CON41-C
CERT C: Rule DCL30-C
CERT C: Rule DCL31-C
CERT C: Rule DCL36-C
CERT C: Rule DCL37-C
CERT C: Rule DCL38-C
CERT C: Rule DCL39-C
CERT C: Rule DCL40-C
CERT C: Rule DCL41-C
CERT C: Rule ENV30-C
CERT C: Rule ENV31-C
CERT C: Rule ENV32-C
CERT C: Rule ENV33-C
CERT C: Rule ENV34-C
CERT C: Rule ERR30-C
CERT C: Rule ERR32-C
CERT C: Rule ERR33-C
CERT C: Rule EXP30-C
CERT C: Rule EXP32-C
CERT C: Rule EXP33-C
CERT C: Rule EXP34-C
CERT C: Rule EXP35-C

CERT C: Rule EXP36-C
CERT C: Rule EXP37-C
CERT C: Rule EXP39-C
CERT C: Rule EXP40-C
CERT C: Rule EXP42-C
CERT C: Rule EXP43-C
CERT C: Rule EXP44-C
CERT C: Rule EXP45-C
CERT C: Rule EXP46-C
CERT C: Rule FIO30-C
CERT C: Rule FIO32-C
CERT C: Rule FIO34-C
CERT C: Rule FIO37-C
CERT C: Rule FIO38-C
CERT C: Rule FIO39-C
CERT C: Rule FIO40-C
CERT C: Rule FIO41-C
CERT C: Rule FIO42-C
CERT C: Rule FIO44-C
CERT C: Rule FIO45-C
CERT C: Rule FIO46-C
CERT C: Rule FIO47-C
CERT C: Rule FLP30-C
CERT C: Rule FLP32-C
CERT C: Rule FLP34-C
CERT C: Rule FLP36-C
CERT C: Rule FLP37-C
CERT C: Rule INT30-C
CERT C: Rule INT31-C
CERT C: Rule INT32-C
CERT C: Rule INT33-C
CERT C: Rule INT34-C
CERT C: Rule INT35-C
CERT C: Rule INT36-C
CERT C: Rule MEM30-C
CERT C: Rule MEM31-C
CERT C: Rule MEM33-C

CERT C: Rule MEM34-C
CERT C: Rule MEM35-C
CERT C: Rule MEM36-C
CERT C: Rule MSC30-C
CERT C: Rule MSC32-C
CERT C: Rule MSC33-C
CERT C: Rule MSC37-C
CERT C: Rule MSC38-C
CERT C: Rule MSC39-C
CERT C: Rule MSC40-C
CERT C: Rule PRE30-C
CERT C: Rule PRE31-C
CERT C: Rule PRE32-C
CERT C: Rule SIG30-C
CERT C: Rule SIG31-C
CERT C: Rule SIG34-C
CERT C: Rule SIG35-C
CERT C: Rule STR30-C
CERT C: Rule STR31-C
CERT C: Rule STR32-C
CERT C: Rule STR34-C
CERT C: Rule STR37-C
CERT C: Rule STR38-C

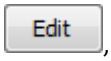
`all`

Check for violations of all CERT C rules and recommendations supported by Polyspace.

See “CERT C Rules and Recommendations”.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click 

, then select the rules and recommendations you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

Dependencies

- This option is available only if you set `Source code language (-lang)` to `C` or `C-CPP`.

  For projects with mixed C and C++ code, the SEI CERT-C checker analyzes only `.c` files.

Command-Line Information

Parameter: `-cert-c`
Value: `all-rules` | `publish-2016` | `all` | `from-file`
Default: `all`
Example (Bug Finder): `polyspace-bug-finder -lang c -sources` *`file_name`* `-cert-c all-rules`
Example (Bug Finder Server): `polyspace-bug-finder-server -lang c -sources` *`file_name`* `-cert-c all-rules`

See Also

`Do not generate results for (-do-not-generate-results-for)`

Topics

“Specify Polyspace Analysis Options”
“Check for Coding Standard Violations”
“CERT C Rules and Recommendations”

Check SEI CERT-C++ (-cert-cpp)

Check for violations of CERT C++ rules

Description

This option affects Bug Finder only.

Specify whether to check for violations of CERT C++ rules.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 1-169 for other options that you must also enable.

Command line: Use the option `-cert-cpp`. See “Command-Line Information” on page 1-170.

Why Use This Option

Use this option to specify the subset of CERT C++ rules to check in your code.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding standard violation, Polyspace assigns a  symbol to the keyword or identifier relevant to the violation.

Settings

Default: `all`

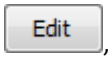
`all`

Check for violations of all CERT C++ rules supported by Polyspace.

See “CERT C++ Rules”.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click 

, then select the rules you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

Dependencies

- This option is available only if you set `Source code language (-lang)` to `CPP` or `C-CPP`.

  For projects with mixed C and C++ code, the SEI CERT-C++ checker analyzes only `.cpp` files.

Command-Line Information

Parameter: `-cert-cpp`
Value: `all` | `from-file` |
Default: `all`
Example (Bug Finder): `polyspace-bug-finder -lang cpp -sources` *`file_name`* `-cert-cpp all`
Example (Bug Finder Server): `polyspace-bug-finder-server -lang cpp -sources` *`file_name`* `-cert-cpp all`

See Also

`Do not generate results for (-do-not-generate-results-for)`

Topics

“Specify Polyspace Analysis Options”
“Check for Coding Standard Violations”

Check ISO/IEC TS 17961 (-iso-17961)

Check for violations of ISO/IEC TS 17961 rules

Description

This option affects Bug Finder only.

Specify whether to check for violations of ISO/IEC TS 17961 rules.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 1-172 for other options that you must also enable.

Command line: Use the option `-iso-17961`. See “Command-Line Information” on page 1-172.

Why Use This Option

Use this option to specify the subset of ISO/IEC TS 17961 rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding standard violation, Polyspace assigns a symbol to the keyword or identifier relevant to the violation.

Settings

Default: `all`

`decidable`

Check for violations of *decidable* rules. Violations of these rules depend only on compile-time static properties, for instance object type or scope of identifiers.

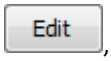
`all`

Check for violations of all ISO/IEC TS 17961 rules Polyspace supports.

See “ISO/IEC TS 17961 Rules”.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click 

, then select the rules and recommendations you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

Dependencies

- This option is available only if you set `Source code language (-lang)` to `C` or `C-CPP`.

Command-Line Information

Parameter: `-iso-17961`
Value:`decidable` | `all` | `from-file`
Default: `all`
Example (Bug Finder): `polyspace-bug-finder -lang c -sources` *`file_name`* `-iso-17961 decidable`

See Also

`Do not generate results for (-do-not-generate-results-for)`

Topics

“Specify Polyspace Analysis Options”
“Check for Coding Standard Violations”
“ISO/IEC TS 17961 Rules”

Calculate code metrics (-code-metrics)

Compute and display code complexity metrics

Description

Specify that Polyspace must compute and display code complexity metrics for your source code. The metrics include file metrics such as number of lines and function metrics such as cyclomatic complexity and estimated size of local variables.

For more information, see “Compute Code Complexity Metrics”.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node.

Command line: Use the option `-code-metrics`. See “Command-Line Information” on page 1-174.

Why Use This Option

By default, Polyspace does not calculate code complexity metrics. If you want these metrics in your analysis results, before running analysis, set this option.

High values of code complexity metrics can lead to obscure code and increase chances of coding errors. Additionally, if you run a Code Prover verification on your source code, you might benefit from checking your code complexity metrics first. If a function is too complex, attempts to verify the function can lead to a lot of unproven code. For information on how to cap your code complexity metrics, see “Compute Code Complexity Metrics”.

Settings

☑ On

Polyspace computes and displays code complexity metrics on the **Results List** pane.

☐ Off (default)

Polyspace does not compute complexity metrics.

Tips

If you want to compute only the code complexity metrics for your code:

- In Bug Finder, disable checking of defects. See `Find defects (-checkers)`.
- In Code Prover, run verification up to the `Source Compliance Checking` phase. See `Verification level (-to)`.

A Code Prover analysis computes the stack usage metrics after the source compliance checking phase. If you stop a Code Prover verification before source compliance checking, the stack usage metrics are not reported.

Command-Line Information

Parameter: `-code-metrics`
Default: Off
Example (Bug Finder): `polyspace-bug-finder -sources file_name -code-metrics`
Example (Code Prover): `polyspace-code-prover -sources file_name -code-metrics`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -code-metrics`
Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -code-metrics`

See Also

Topics

“Compute Code Complexity Metrics”
“Code Metrics”

Find defects (-checkers)

Enable or disable defect checkers

Description

This option affects a Bug Finder analysis only.

Enable checkers for bugs/coding defects.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Bug Finder Analysis** node.

Command line: Use the option `-checkers`. See “Command-Line Information” on page 1-176.

Why Use This Option

The default set of checkers is designed to find the most meaningful bugs in most software development situations. If you have specific needs, enable or disable individual defect checkers. For instance, if you want to follow a specific security standard, choose a different subset of checkers.

Settings

Default: `default`

`default`

A subset of defects defined by the software.

See “Polyspace Bug Finder Defects Checkers Enabled by Default”.

`all`

All defects.

For a list of all defects checkers, see Bug Finder Defects.

`CWE`

A subset of defects that correspond to CWE™ IDs.

See “CWE Coding Standard and Polyspace Results”.

`custom`

Choose the defects you want to find by selecting categories of checkers or specific defects.

Tips

You can use a spreadsheet to keep track of the defect checkers that you enable and add notes explaining why you do not enable the other checkers. A spreadsheet of checkers is provided in *`polyspaceroot`*`\polyspace\resources`. Here, *`polyspaceroot`* is the Polyspace installation folder, such as `C:\Program Files\Polyspace\R2019a`.

Command-Line Information

Regardless of order, the shell script processes the `-checkers` option, and then `-disable-checkers` option.

For the command-line parameters values, see “Short Names of Bug Finder Defect Checkers”.
Parameter: `-checkers`
Value: `default | all | none | CWE |defect group | defect parameters`
Default: `default`
Parameter: `-disable-checkers`
Value: `defect group |defect parameters`
Example 1 (Bug Finder): `polyspace-bug-finder -sources filename -checkers numerical,data_flow -disable-checkers FLOAT_ZERO_DIV`
Example 2 (Bug Finder): `polyspace-bug-finder -sources filename -checkers default -disable-checkers concurrency,dead_code`
Example 1 (Bug Finder Server): `polyspace-bug-finder-server -sources filename -checkers numerical,data_flow -disable-checkers FLOAT_ZERO_DIV`
Example 2 (Bug Finder Server): `polyspace-bug-finder-server -sources filename -checkers default -disable-checkers concurrency,dead_code`

See Also

“Defects”

Topics

“Specify Polyspace Analysis Options”
“Short Names of Bug Finder Defect Checkers”
“Bug Finder Defect Groups”

Run stricter checks considering all values of system inputs (-checks-using-system-input-values)

Enable stricter checks and provide examples of values that lead to detected defect

Description

This option affects a Bug Finder analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Enable a stricter static analysis for a subset of numerical and static memory defect checkers. Polyspace considers all possible values of system inputs and provides examples of values that can lead to detected defects. For each function `foo` that you specify with `Consider inputs to these functions (-system-inputs-from)`, these are the system inputs.

- Each argument of `foo`.
- Each read of a global variable by `foo` or one of its callees.

  For the `main()` function, the analysis assumes that the global variables are initialized with value 0.
- Each read of a volatile variable by `foo` or one of its callees.
- Each return of a stubbed function. a Bug Finder analysis stubs a function if you do not provide the body of the function in your source code.

The stricter checks are enabled for this subset of defect checkers.

Defect checkers subset

- `Array access out of bounds`
- `Bitwise operation on negative value`
- `Float conversion overflow`
- `Float overflow`
- `Float division by zero`
- `Integer conversion overflow`
- `Integer division by zero`
- `Integer overflow`
- `Invalid use of standard library floating point routine`
- `Invalid use of standard library integer routine`
- `Null pointer`
- `Shift of a negative value`
- `Shift operation overflow`
- `Sign change integer conversion overflow`

- `Unsigned integer conversion overflow`
- `Unsigned integer overflow`

You can view examples of values that lead to the detected defects in the **Events** column of the **Results Details** pane on the desktop interface or the Polyspace Access web interface.

Set Option

User interface (desktop products only): In the **Configuration** pane, the option is on the **Bug Finder Analysis** node. See “Dependencies” on page 1-178 for other options that you must also enable.

Command line: Use the option `-checks-using-system-input-values`. See “Command-Line Information” on page 1-179.

Why Use This Option

The default Bug Finder analysis does not flag defects that are caused by specific values of unknown inputs. Since the inputs might be bounded or initialized in a source file that you are not analyzing, or the specific value causing a defect might not occur in practice, the default analysis behavior helps to minimize false positives.

Enable this option to run a stricter analysis on a function whose system inputs might cause sporadic run-time errors during execution. Using this option might result in a longer analysis time.

Settings

☑ On

Polyspace considers all possible values of system inputs for a subset of numerical and static memory defect checkers and provides examples of values that lead to detected defects.

○ Off (default)

Polyspace considers possible values of a system input only if the input is bounded by constraints in your code such as `assert` or `if`. The analysis provides no examples of values that lead to detected defects.

Dependencies

- In the desktop interface, this option is enabled only if you enable `Find defects (-checkers)`.
- This option is ignored if you enable `Use fast analysis mode for Bug Finder (-fast-analysis)`.

Tips

- If you set external constraints on global variables, the analysis shows examples of global variable values causing defects only within these constraints. See `Constraint setup (-data-range-specifications)`.
- If the input is a pointer `p`, the analysis assumes that the pointer is not null and can be safely dereferenced. The example value of the input causing a defect is the value of `*p`. This value is represented as an array in the **Results Details** pane. For instance, in this code snippet:

```
void func(int* x){
    int tmp= *(x+3);
```

```
    if(1/(tmp-4))
        return;
}
```

The example value of the input causing a defect is {0,0,0,4}, where the array represents *x, *(x+1), *(x+2), and *(x+3). The value *(x+3)=4 causes a division by zero.

- The analysis treats these standard library functions that read values from external sources as stubbed functions.
  - getchar
  - getc
  - fgetc
  - scanf
- The stricter analysis considers all possible values of system inputs but it is not an exhaustive analysis. If Bug Finder cannot determine whether a particular input causes a defect, no defect is shown. For more on exhaustive analysis, see "Choose Between Polyspace Bug Finder and Polyspace Code Prover".

Command-Line Information

Parameter: -checks-using-system-input-values
Default: Off
Example (Bug Finder): polyspace-bug-finder -sources *file_name* -checkers numerical,static_memory -checks-using-system-input-values
Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -checkers numerical,static_memory -checks-using-system-input-values

See Also

Array access out of bounds | Bitwise operation on negative value | Consider inputs to these functions (-system-inputs-from) | Float conversion overflow | Float division by zero | Float overflow | Integer overflow | Integer conversion overflow | Integer division by zero | Invalid use of standard library floating point routine | Invalid use of standard library integer routine | Null pointer | Shift of a negative value | Shift operation overflow | Sign change integer conversion overflow | Unsigned integer overflow | Unsigned integer conversion overflow

Topics

"Specify Polyspace Analysis Options"
"Extend Bug Finder Checkers to Find Defects from Specific System Input Values"

Introduced in R2020a

Consider inputs to these functions (`-system-inputs-from`)

Specify functions for which the analysis considers all possible input values

Description

This option affects a Bug Finder analysis only.

Specify the functions in your code for which Polyspace considers all possible input values. For each function that you specify with this option, the analysis considers all possible values of these inputs:

- Each argument of the function.
- Each read of a global variable by the function or one of its callees.

  For the `main()` function, the analysis assumes that the global variables are initialized with value 0.
- Each read of a volatile variable by the function or one of its callees.
- Each return of a stubbed function. a Bug Finder analysis stubs a function if you do not provide the body of the function in your source code.

Set Option

User interface (desktop products only): In the **Configuration** pane, the option is on the **Bug Finder Analysis** node. See “Dependencies” on page 1-181 for other options that you must also enable.

Command line: Use the option `-system-inputs-from`. See “Command-Line Information” on page 1-181.

Why Use This Option

By default, Polyspace considers all possible input values for the `main()` function and tasks, if any, or uncalled functions with at least one callee if your code has no `main()`. Depending on the issue that you are investigating by running the stricter checks, specify a different subset of functions to analyze.

Settings

Default: `auto`

`auto`

Consider all possible values for inputs to `main()` function and tasks, if any. You specify tasks with these options.

- `Cyclic tasks (-cyclic-tasks)`
- `Tasks (-entry-points)`
- `Interrupts (-interrupts)`

When the analyzed code has no `main()`, the analysis considers all possible values for inputs to uncalled functions with at least one callee.

`uncalled`

Consider all possible values for inputs to all uncalled functions.

`all`

Consider all possible values for inputs to all functions.

`custom`

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

Dependencies

This option is enabled only if you enable `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`.

Tips

- The analysis treats these standard library functions that read values from external sources as stubbed functions.
  - `getchar`
  - `getc`
  - `fgetc`
  - `scanf`

Command-Line Information

Parameter: `-system-inputs-from`
Value: `auto | uncalled | all | custom`
Default: `auto`
Example (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-checks-using-system-input-values -system-inputs-from custom=`*`func1`*`,`*`func2`*
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-checks-using-system-input-values -system-inputs-from custom=`*`func1`*`,`*`func2`*

See Also

`Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`

Topics
“Specify Polyspace Analysis Options”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2020a

Class (-class-analyzer)

Specify classes that you want to verify

Description

This option affects a Code Prover analysis only.

Specify classes that Polyspace uses to generate a `main`.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-182 for other options that you must also enable.

Command line: Use the option `-class-analyzer`. See “Command-Line Information” on page 1-183.

Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option and the option `Functions to call within the specified classes (-class-analyzer-calls)` to specify the class methods that the generated `main` must call. Unless a class method is called directly or indirectly from `main`, the software does not analyze the method.

Settings

Default: `all`

`all`

Polyspace can use all classes to generate a `main`. The generated `main` calls methods that you specify using **Functions to call within the specified classes**.

`none`

The generated `main` cannot call any class method.

`custom`

Polyspace can use classes that you specify to generate a `main`. The generated `main` calls methods from classes that you specify using **Functions to call within the specified classes**.

Dependencies

You can use this option only if all of the following are true:

- Your code does not contain a `main` function.
- `Source code language (-lang)` is set to `CPP` or `C-CPP`.
- `Verify module or library (-main-generator)` is selected.

Tips

If you select none for this option, Polyspace will not verify class methods that you do not call explicitly in your code.

Command-Line Information

Parameter: -class-analyzer
Value: all | none | custom=*class1*[,*class2*,...]
Default: all
Example (Code Prover): polyspace-code-prover -sources *file_name* -main-generator -class-analyzer custom=myClass1,myClass2
Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -main-generator -class-analyzer custom=myClass1,myClass2

See Also

Analyze class contents only (-class-only) | Functions to call within the specified classes (-class-analyzer-calls) | Skip member initialization check (-no-constructors-init-check) | Verify module or library (-main-generator)

Topics

“Specify Polyspace Analysis Options”
“Verify C++ Classes” (Polyspace Code Prover)

Functions to call within the specified classes (-class-analyzer-calls)

Specify class methods that you want to verify

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify class methods that Polyspace uses to generate a `main`. The generated `main` can call static, public and protected methods in classes that you specify using the **Class** option.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-185 for other options that you must also enable.

Command line: Use the option `-class-analyzer-calls`. See “Command-Line Information” on page 1-185.

Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option and the option `Class (-class-analyzer)` to specify the class methods that the generated `main` must call. Unless a class method is called directly or indirectly from `main`, the software does not analyze the method.

Settings

Default: `unused`

`all`

The generated `main` calls all public and protected methods. It does not call methods inherited from a parent class.

`all-public`

The generated `main` calls all public methods. It does not call methods inherited from a parent class.

`inherited-all`

The generated `main` calls all public and protected methods including those inherited from a parent class.

`inherited-all-public`

The generated `main` calls all public methods including those inherited from a parent class.

`unused`

The generated `main` calls public and protected methods that are not called in the code.

`unused-public`

The generated `main` calls public methods that are not called in the code. It does not call methods inherited from a parent class.

`inherited-unused`

The generated `main` calls public and protected methods that are not called in the code including those inherited from a parent class.

`inherited-unused-public`

The generated `main` calls public methods that are not called in the code including those inherited from a parent class.

`custom`

The generated `main` calls the methods that you specify.

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

Dependencies

You can use this option only if:

- `Source code language (-lang)` is set to `CPP` or `C-CPP`.
- `Verify module or library (-main-generator)` is selected.

Command-Line Information

Parameter: `-class-analyzer-calls`
Value: `all` | `all-public` | `inherited-all` | `inherited-all-public` | `unused` | `unused-public` | `inherited-unused` | `inherited-unused-public` | `custom=`*`method1`*`[,`*`method2`*`,...]`
Default: `unused`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public`

See Also

`Class (-class-analyzer)` | `Verify module or library (-main-generator)`

Topics
“Specify Polyspace Analysis Options”

“Verify C++ Classes” (Polyspace Code Prover)

Analyze class contents only (-class-only)

Do not analyze code other than class methods

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify that Polyspace must verify only methods of classes that you specify using the option `Class (-class-analyzer)`.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-187 for other options that you must also enable.

Command line: Use the option `-class-only`. See “Command-Line Information” on page 1-188.

Why Use This Option

Use this option to restrict the analysis to certain class methods only.

You specify these methods through the options:

- `Class (-class-analyzer)`
- `Functions to call within the specified classes (-class-analyzer-calls)`

When you analyze a module or library, Code Prover generates a `main` function if one does not exist. The `main` function calls class methods using these two options and functions that are not class methods using other options. Code Prover analyzes these methods and functions for robustness to all inputs. If you use this option, Code Prover analyzes the methods only.

Settings

☑ On

Polyspace verifies the class methods only. It stubs functions out of class scope even if the functions are defined in your code.

☐ Off (default)

Polyspace verifies functions out of class scope in addition to class methods.

Dependencies

You can use this option only if all of the following are true:

- Your code does not contain a `main` function.
- `Source code language (-lang)` is set to `CPP` or `C-CPP`.

- `Verify module or library (-main-generator)` is selected.

If you select this option, you must specify the classes using the `Class (-class-analyzer)` option.

Tips

Use this option:

- For robustness verification of class methods. Unless you use this option, Polyspace verifies methods that you call in your code only for your input combinations.
- In case of scaling.

Command-Line Information

Parameter: `-class-only`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public -class-only`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public -class-only`

See Also

`Class (-class-analyzer)` | `Functions to call within the specified classes (-class-analyzer-calls)` | `Verify module or library (-main-generator)`

Topics
“Specify Polyspace Analysis Options”
“Verify C++ Classes” (Polyspace Code Prover)

Initialization functions (-functions-called-before-main)

Specify functions that you want the generated main to call ahead of other functions

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify functions that you want the generated main to call ahead of other functions.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-190 for other options that you must also enable.

Command line: Use the option -functions-called-before-main. See “Command-Line Information” on page 1-190.

Why Use This Option

If you are verifying a module or library, Code Prover generates a main function if one does not exist. If a main exists, the analysis uses the existing main.

Use this option along with the option Functions to call (-main-generator-calls) to specify which functions the generated main must call. Unless a function is called directly or indirectly from main, the software does not analyze the function.

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

If the function or method is not overloaded, specify the function name. Otherwise, specify the function prototype with arguments. For instance, in the following code, you must specify the prototypes func(int) and func(double).

```
int func(int x) {
 return(x * 2);
}
double func(double x) {
 return(x * 2);
}
```

For C++, if the function is:

- A class method: The generated `main` calls the class constructor before calling this function.
- Not a class method: The generated `main` calls this function before calling class methods.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::init(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::init()`.

Dependencies

This option is enabled only if you select **Verify module or library** under **Code Prover Verification** and your code does not contain a `main` function.

Tips

Although these functions are called ahead of other functions, they can be called in arbitrary order. If you want to call your initialization functions in a specific order, manually write a `main` function to call them.

Command-Line Information

Parameter: `-functions-called-before-main`
Value: *`function1`*`[,`*`function2`*`[,...]]`
No Default
Example 1 (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-main-generator -functions-called-before-main myfunc`
Example 2 (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-main-generator -functions-called-before-main myClass::init(int)`
Example 1 (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -functions-called-before-main myfunc`
Example 2 (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -functions-called-before-main myClass::init(int)`

See Also

`Class (-class-analyzer)` | `Functions to call (-main-generator-calls)` | `Functions to call within the specified classes (-class-analyzer-calls)` | `Variables to initialize (-main-generator-writes-variables)` | `Verify module or library (-main-generator)`

Topics

“Specify Polyspace Analysis Options”
“Verify C Application Without main Function” (Polyspace Code Prover)
“Verify C++ Classes” (Polyspace Code Prover)

Verify initialization section of code only (-init-only-mode)

Check initialization code alone for run-time errors and other issues

Description

This option affects a Code Prover analysis only.

Specify that Polyspace must check only the section of code marked as initialization code for run-time errors and other issues.

To indicate the end of initialization code, you enter the line

```
#pragma polyspace_end_of_init
```

in the `main` function (only once). The initialization code starts from the beginning of `main` and continues up to this pragma.

Since compilers ignore unrecognized pragmas, the presence of this pragma does not affect program execution.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node.

Command line: Use the option `-init-only-mode`. See “Command-Line Information” on page 1-193.

Why Use This Option

Often, issues in the initialization code can invalidate the analysis of the remaining code. You can use this option to check the initialization code alone and fix the issues, and then disable this option to verify the remaining program.

For instance, in this example:

```
#include <limits.h>

int aVar;
const int aConst = INT_MAX;
int anotherVar;

int main() {
      aVar = aConst + 1;
#pragma polyspace_end_of_init
      anotherVar = aVar - 1;
      return 0;
}
```

the overflow in the line `aVar = aConst+1` must be fixed first before the value of `aVar` is used in subsequent code.

Settings

☑ On

Polyspace checks the code from the beginning of `main` and continues up to the pragma `polyspace_end_of_init`.

○ Off (default)

Polyspace checks the complete application beginning from the `main` function.

Dependencies

You can use this option and designate a section of code as initialization code only if:

- Your program contains a `main` function and you use the option `Verify whole application` (implicitly set by default at command line).
- You set `Source code language (-lang)` to `C`.

Note that the pragma must appear only once in the `main` function. The pragma can appear before or after variable declarations but must appear after type definitions (`typedef`-s).

You cannot use this option with the following options:

- `Verify files independently (-unit-by-unit)`
- `Show global variable sharing and usage only (-shared-variables-mode)`

Tips

- Use this option along with the option `Check that global variables are initialized after warm reboot (-check-globals-init)` to thoroughly check the initialization code before checking the remaining program. If you use both options, the verification checks for the following:
  - Definite or possible run-time errors in the initialization code.
  - Whether all non-`const` global variables are initialized along all execution paths through the initialization code.
- Multitasking options are disabled if you check initialization code only because the initialization of global variables is expected to happen before the tasks (threads) begin. As a result, task bodies are not verified.

  See also “Multitasking”.
- If you check initialization code only, the analysis truncates execution paths containing the pragma at the location of the pragma but continues to check other execution paths.

  For instance, in this example, the `pragma` appears in an `if` block. A red non-initialized variable check appears on the line `int a = var` because the path containing the initialization stops at the location of the pragma. On the only other remaining path that bypasses the `if` block, the variable `var` is not initialized.

  ```
  int var;

  int func();
  ```

```
int main() {
    int err = func();
    if(err) {
        var = 0;
 #pragma polyspace_end_of_init
    }
    int a = var;
    return 0;
}
```

To avoid these situations, try to place the pragma outside a block. See other suggestions for placement of the pragma in the reference for `Check that global variables are initialized after warm reboot (-check-globals-init)`.

- To determine the initialization of a structure, a regular Code Prover analysis only considers fields that are used.

  If you check initialization code only using this option, the analysis covers only a portion of the code and cannot determine if a variable is used beyond this portion. Therefore, the checks for initialization consider all structure fields, whether used or not.

Command-Line Information

Parameter: `-init-only-mode`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-init-only-mode`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-init-only-mode`

See Also

`Check that global variables are initialized after warm reboot (-check-globals-init)`

Topics

“Specify Polyspace Analysis Options”

Introduced in R2020a

Verify whole application

Stop verification if sources files are incomplete and do not contain a `main` function

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify that Polyspace verification must stop if a `main` function is not present in the source files.

If you select a Visual C++ setting for `Compiler (-compiler)`, you can specify which function must be considered as `main`. See `Main entry point (-main)`.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node.

Command line: There is no corresponding command-line option. See “Command-Line Information” on page 1-194.

Settings

On

Polyspace verification stops if it does not find a `main` function in the source files.

Off (default)

Polyspace continues verification even when a `main` function is not present in the source files. If a `main` is not present, it generates a file `__polyspace_main.c` that contains a `main` function.

Tips

If you use this option, your code must contain a `main` function. Otherwise you see the error:

```
Error: required main procedure not found
```

If your code does not contain a `main` function, use the option `Verify module or library (-main-generator)` to generate a `main` function.

Command-Line Information

Unlike the user interface, by default, a verification from the command line stops if it does not find a `main` function in the source files. If you specify the option `-main-generator`, Polyspace generates a `main` if it cannot find one in the source files.

See Also

`Show global variable sharing and usage only (-shared-variables-mode)` | `Verify module or library (-main-generator)`

Topics
“Specify Polyspace Analysis Options”
“Verify C Application Without main Function” (Polyspace Code Prover)
“Verify C++ Classes” (Polyspace Code Prover)

Show global variable sharing and usage only (-shared-variables-mode)

Compute global variable sharing and usage without running full analysis

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify this option to run a less extensive analysis that computes the global variable sharing and usage in your entire application. The analysis does not verify your code for run-time errors. The analysis results also include coding standards violations if you enable coding standards checking, and code metrics if you enable code metrics computation.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node.

Command line: Use the option `-shared-variables-mode`. See “Command-Line Information” on page 1-197.

Why Use This Option

You can see global variable sharing and usage without running a full analysis on your entire application that includes run-time error detection. Run-time error detection on an entire application can take a long time.

Settings

☑ On

Polyspace computes global variable sharing and usage but does not verify your code for run-time errors.

○ Off (default)

Polyspace runs a full analysis on your code, including run-time error detection.

Dependencies

- You can use this option only if your program contains a `main` function and you enable the option `Verify whole application` (implicitly set by default at command line).
- When you enable this option, you must also enable at least one of these options.
  -
  -
  -

-
-
-

Tips

- After you analyze your complete application to see global variable sharing and usage, run a component-by-component Code Prover analysis to detect run-time errors.
- In the desktop product, you can see all read and write operations on global variables in the "Variable Access" (Polyspace Code Prover) pane.
- In this less extensive analysis mode, the analysis checks for most but not all coding standards violations, and computes most but not all code metrics.

Command-Line Information

Parameter: `-shared-variables-mode`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-shared-variables-mode -enable-concurrency-detection`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-shared-variables-mode -enable-concurrency-detection`

See Also

Topics
"Specify Polyspace Analysis Options"

Introduced in R2019b

Main entry point (-main)

Specify a Microsoft Visual C++ extensions of main

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify the function that you want to use as main. If the function does not exist, the verification stops with an error message. Use this option to specify Microsoft Visual C++ extensions of main.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-198 for other options that you must also enable.

Command line: Use the option -main. See “Command-Line Information” on page 1-199.

Settings

Default: _tmain

_tmain
: Use _tmain as entry point to your code.

wmain
: Use wmain as entry point to your code.

_tWinMain
: Use _tWinMain as entry point to your code.

wWinMain
: Use wWinMain as entry point to your code.

WinMain
: Use WinMain as entry point to your code.

DllMain
: Use DllMain as entry point to your code.

Dependencies

This option is enabled only if you:

- Set Source code language (-lang) to CPP.
- Select Verify whole application.

Command-Line Information

Parameter: -main
Value: _tmain | wmain | _tWinMain | wWinMain | WinMain | DllMain
Example (Code Prover): polyspace-code-prover -sources *file_name* -compiler visual14.0 -main _tmain
Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -compiler visual14.0 -main _tmain

See Also

Verify module or library (-main-generator) | Verify whole application

Topics

“Specify Polyspace Analysis Options”

Functions to call (`-main-generator-calls`)

Specify functions that you want the generated `main` to call after the initialization functions

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify functions that you want the generated `main` to call. The `main` calls these functions after the ones you specify through the option `Initialization functions (-functions-called-before-main)`.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-201 for other options that you must also enable.

Command line: Use the option `-main-generator-calls`. See “Command-Line Information” on page 1-201.

Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option along with the option `Initialization functions (-functions-called-before-main)` to specify which functions the generated `main` must call. Unless a function is called directly or indirectly from `main`, the software does not analyze the function.

Settings

Default: `unused`

`none`

The generated `main` does not call any function.

`unused`

The generated `main` calls only those functions that are not called in the source code. It does not call inlined functions.

`all`

The generated `main` calls all functions except inlined ones.

`custom`

The generated `main` calls functions that you specify.

Enter function names or choose from a list.

- Click  to add a field and enter the function name.

- Click  to list functions in your code. Choose functions from the list.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

Dependencies

This option is available only if you select `Verify module or library (-main-generator)`.

Tips

- Select `unused` when you use **Code Prover Verification** > **Verify files independently**.
- If you want the generated `main` to call an inlined function, select `custom` and specify the name of the function.
- To verify a multitasking application without a main, select `none`.
- The generated `main` can call the functions in arbitrary order. If you want to call your functions in a specific order, manually write a `main` function to call them.
- To specify instantiations of templates as arguments, run analysis once with the option argument `all`. Search for the template name in the analysis log and use the template name as it appears in the analysis log for the option argument.

  For instance, to specify this template function instantiation as option argument:

  ```
  template <class T>
  T GetMax (T a, T b) {
    T result;
    result = (a>b)? a : b;
    return (result);
  }
  template int GetMax<int>(int, int); // explicit instantiation
  ```

  Run an analysis with the option `-main-generator-calls all`. Search for `getMax` in the analysis log. You see the function format:

  ```
  T1 getMax<int>(T1, T1)
  ```

  To call only this template instantiation, remove the space between the arguments and use the option:

  ```
  -main-generator-calls custom="T1 getMax<int>(T1,T1)"
  ```

Command-Line Information

Parameter: `-main-generator-calls`
Value: `none` | `unused` | `all` | `custom=`*`function1`*`[,`*`function2`*`[,...]]`
Default: `unused`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-main-generator -main-generator-calls all`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -main-generator-calls all`

See Also

`Class (-class-analyzer)` | `Functions to call within the specified classes (-class-analyzer-calls)` | `Initialization functions (-functions-called-before-main)` | `Verify module or library (-main-generator)`

Topics

“Specify Polyspace Analysis Options”
“Verify C Application Without main Function” (Polyspace Code Prover)

Variables to initialize (-main-generator-writes-variables)

Specify global variables that you want the generated main to initialize

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify global variables that you want the generated main to initialize. Polyspace considers these variables to have any value allowed by their type.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-204 for other options that you must also enable.

Command line: Use the option -main-generator-writes-variables. See “Command-Line Information” on page 1-204.

Why Use This Option

If you are verifying a module or library, Code Prover generates a main function if one does not exist. If a main exists, the analysis uses the existing main.

Use this option to specify which global variables the generated main must initialize.

Settings

Default:

- C code — public
- C++ Code — uninit

uninit
: C++ Only

  The generated main only initializes global variables that you have not initialized during declaration.

none
: The generated main does not initialize global variables.

public
: The generated main initializes all global variables except those declared with keywords static and const.

all
: The generated main initializes all global variables except those declared with keyword const.

`custom`

The generated `main` only initializes global variables that you specify. Click  to add a field. Enter a global variable name.

Dependencies

You can use this option only if the following are true:

- Your code does not contain a `main` function.
- `Verify module or library (-main-generator)` is selected.

The option is disabled if you enable the option `Ignore default initialization of global variables (-no-def-init-glob)`. Global variables are considered as uninitialized until you explicitly initialize them in the code.

Tips

This option only affects global variables that are defined in the project. If a global variable is declared as `extern`, the analysis considers that the variable can have any value allowed by its data type, irrespective of the value of this option.

Command-Line Information

Parameter: `-main-generator-writes-variables`
Value: `uninit | none | public | all | custom=`*`variable1`*`[,`*`variable2`*`[,...]]`
Default: (C) `public` | (C++) `uninit`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-main-generator -main-generator-writes-variables all`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -main-generator-writes-variables all`

See Also

`Verify module or library (-main-generator)`

Topics
“Specify Polyspace Analysis Options”
“Verify C Application Without main Function” (Polyspace Code Prover)

Skip member initialization check (-no-constructors-init-check)

Do not check if class constructor initializes class members

Description

This option affects a Code Prover analysis only.

Specify that Polyspace must not check whether each class constructor initializes all class members.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-205 for other options that you must also enable.

Command line: Use the option `-no-constructors-init-check`. See “Command-Line Information” on page 1-206.

Why Use This Option

Use this option to disable checks for initialization of class members in constructors.

Settings

☑ On

Polyspace does not check whether each class constructor initializes all class members.

☐ Off (default)

Polyspace checks whether each class constructor initializes all class members. It uses the functions `check_NIV()` and `check_NIP()` in the generated `main` to perform these checks. It checks for initialization of:

- Integer types such as `int`, `char` and `enum`, both `signed` or `unsigned`.
- Floating-point types such as `float` and `double`.
- Pointers.

Dependencies

You can use this option only if all of the following are true:

- Your code does not contain a `main` function.
- `Source code language (-lang)` is set to `CPP` or `C-CPP`.
- `Verify module or library (-main-generator)` is selected.

If you select this option, you must specify the classes using the`Class (-class-analyzer)` option.

Command-Line Information

Parameter: `-no-constructors-init-check`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources file_name -main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public -no-constructors-init-check`
Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public -no-constructors-init-check`

See Also

`Class (-class-analyzer)` | `Verify module or library (-main-generator)`

Topics

“Specify Polyspace Analysis Options”
“Verify C++ Classes” (Polyspace Code Prover)

Verify files independently (-unit-by-unit)

Verify each source file independently of other source files

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify that each source file must be verified independently of other source files. Each file is verified individually, independent of other files in the module. Verification results can be viewed for the entire project or for individual files.

After you open the verification result for one file, in the user interface of the Polyspace desktop products, you can see a summary of results for all files on the **Dashboard** pane. You can open the results for each file directly from this summary table.

Each result file (with name `ps_results.pscp`) is saved in a subfolder of the results folder. The subfolder has the same name as the source file being analyzed.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-207 for other options that you must also enable.

Command line: Use the option `-unit-by-unit`. See “Command-Line Information” on page 1-208.

Why Use This Option

There are many reasons you might want to verify each source file independently of other files.

For instance, if verification of a project takes very long, you can perform a file by file verification to identify which file is slowing the verification.

Settings

☑ On

Polyspace creates a separate verification job for each source file.

☐ Off (default)

Polyspace creates a single verification job for all source files in a module.

Dependencies

This option is enabled only if you select `Verify module or library (-main-generator)`.

Tips

- Code Prover requires a `main` function as the starting point of verification. In the file-by-file mode, because most files do not have a `main`, Code Prover generates a `main` function when required. By default, the generated `main` calls uncalled functions (uncalled non-private methods and out-of-class functions in C++). For more information, see:
  - "Verify C Application Without main Function" (Polyspace Code Prover)
  - "Verify C++ Classes" (Polyspace Code Prover)
- If you perform a file by file verification, you cannot specify multitasking options.
- If your verification for the entire project takes very long, perform a file by file verification. After the verification is complete for a file, you can view the results while other files are still being verified.
- You can generate a report of the verification results for each file or for all the files together. To generate a single report for all files, perform the report generation after verification (and not along with verification using analysis options).

  To generate a single report for all the files in the Polyspace user interface (desktop product only):

  1. Open the results for one file.
  2. Select **Reporting > Run Report**. Before generating the report, select the option **Generate a single report including all unit results**.
- When you perform a file-by-file verification, you can see many instances of unused variables. Some of these variables might be used in other files but show as unused in a file-by-file verification.

  If you want to ignore these results, use a review scope (named set of filters) that filters out unused variables. See "Filter and Group Results".

Command-Line Information

Parameter: `-unit-by-unit`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources file_name -unit-by-unit`
Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -unit-by-unit`

See Also

`Common source files (-unit-by-unit-common-source)`

Topics
"Specify Polyspace Analysis Options"

Common source files (-unit-by-unit-common-source)

Specify files that you want to include with each source file during a file by file verification

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

For a file by file verification, specify files that you want to include with each source file verification. These files are compiled once, and then linked to each verification.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 1-209 for other options that you must also enable.

Command line: Use the option `-unit-by-unit-common-source`. See “Command-Line Information” on page 1-210.

Why Use This Option

There are many reasons you might want to verify each source file independently of other files. For instance, if verification of a project takes very long, you can perform a file by file verification to identify which file is slowing the verification.

If you perform a file by file verification, some of your files might be missing information present in the other files. Place the missing information in a common file and use this option to specify the file for verification. For instance, if multiple source files call the same function, use this option to specify a file that contains the function definition or a function stub. Otherwise, Polyspace uses its own stubs for functions that are called but not defined in the source files. The assumptions behind the Polyspace stubs can be broader than what you want, leading to orange checks.

Settings

No Default

Click  to add a field. Enter the full path to a file. Otherwise, use the  button to navigate to the file location.

Dependencies

This option is enabled only if you select `Verify files independently (-unit-by-unit)`.

Command-Line Information

Parameter: `-unit-by-unit-common-source`
Value: *`file1`*`[,`*`file2`*`[,...]]`
No Default
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-unit-by-unit -unit-by-unit-common-source definitions.c`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-unit-by-unit -unit-by-unit-common-source definitions.c`

See Also

`Verify files independently (-unit-by-unit)`

Topics

“Specify Polyspace Analysis Options”

Verify model generated code (-main-generator)

Specify that a main function must be generated if it is not present in source files

Description

In Bug Finder, use this option only for code generated from MATLAB code or Simulink models.

Specify that Polyspace must generate a main function if it does not find one in the source files.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node.

Command line: Use the option -main-generator. See “Command-Line Information” on page 1-211.

Settings

This option is always enabled for code generated from models.

Polyspace generates a main function for the analysis. The generated main contains cyclic code that executes in a loop. The loop can run an unspecified number of times.

The main performs the following functions before the loop begins:

- Initializes variables specified by Parameters (-variables-written-before-loop).
- Calls the functions specified by Initialization functions (-functions-called-before-loop).

The main then performs the following functions in the loop:

- Calls the functions specified by Step functions (-functions-called-in-loop).
- Writes to variables specified by Inputs (-variables-written-in-loop).

Finally, the main calls the functions specified by Termination functions (-functions-called-after-loop).

Command-Line Information

Parameter: -main-generator
Default: On
Example (Bug Finder): polyspace-bug-finder -sources *file_name* -main-generator ...
Example (Code Prover): polyspace-code-prover -sources *file_name* -main-generator ...
Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -main-generator ...
Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -main-generator ...

See Also

`Initialization functions (-functions-called-before-loop)` | `Inputs (-variables-written-in-loop)` | `Parameters (-variables-written-before-loop)` | `Step functions (-functions-called-in-loop)` | `Termination functions (-functions-called-after-loop)` | `Verify model generated code (-main-generator)`

Topics

“Configure Advanced Polyspace Options in Simulink”
“How Polyspace Analysis of Generated Code Works”

Initialization functions (-functions-called-before-loop)

Specify functions that the generated `main` must call before the cyclic code loop

Description

Use this option only for code generated from MATLAB code or Simulink models.

Specify functions that the generated `main` must call before the cyclic code begins.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node.

Command line: Use the option `-functions-called-before-loop`. See “Command-Line Information” on page 1-213.

Settings

No Default

Click  to add a field. Enter function name.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::init(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::init()`.

Tips

- If you specify a function for the option `Termination functions (-functions-called-after-loop)`, you cannot specify it for this option.

Command-Line Information

Parameter: `-functions-called-before-loop`
No Default
Value: *`function1`*`[,`*`function2`*`[,...]]`
Example (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-main-generator -functions-called-before-loop myfunc`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-main-generator -functions-called-before-loop myfunc`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-main-generator -functions-called-before-loop myfunc`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -functions-called-before-loop myfunc`

See Also

`Step functions (-functions-called-in-loop)` | `Termination functions (-functions-called-after-loop)` | `Verify model generated code (-main-generator)`

Topics

“Configure Advanced Polyspace Options in Simulink”
“How Polyspace Analysis of Generated Code Works”

Step functions (-functions-called-in-loop)

Specify functions that the generated main must call in the cyclic code loop

Description

Use this option only for code generated from MATLAB code or Simulink models.

Specify functions that the generated main must call in each cycle of the cyclic code.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node.

Command line: Use the option -functions-called-in-loop. See “Command-Line Information” on page 1-215.

Settings

Default: none

none

The generated main does not call functions in the cyclic code.

all

The generated main calls all functions except inlined ones. If you specify certain functions for the options **Initialization functions** or **Termination functions**, the generated main does not call those functions in the cyclic code.

custom

The generated main calls functions that you specify. Click  to add a field. Enter function name.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, myClass::myMethod(int). If the function does not have a parameter, use an empty parenthesis, for instance, myClass::myMethod().

Tips

If you have specified a function for the option Initialization functions (-functions-called-before-loop) or Termination functions (-functions-called-after-loop), to call it inside the cyclic code, use custom and specify the function name.

Command-Line Information

Parameter: -functions-called-in-loop
Value: none | all | custom=*function1*[,*function2*[,...]]
Default: none
Example (Bug Finder): polyspace-bug-finder -sources *file_name* -main-generator -functions-called-in-loop all

Example (Code Prover): `polyspace-code-prover -sources file_name -main-generator -functions-called-in-loop all`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources file_name -main-generator -functions-called-in-loop all`
Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -main-generator -functions-called-in-loop all`

See Also

`Initialization functions (-functions-called-before-loop)` | `Termination functions (-functions-called-after-loop)` | `Verify model generated code (-main-generator)`

Topics

“Configure Advanced Polyspace Options in Simulink”
“How Polyspace Analysis of Generated Code Works”

Termination functions (-functions-called-after-loop)

Specify functions that the generated `main` must call after the cyclic code loop

Description

Use this option only for code generated from MATLAB code or Simulink models.

Specify functions that the generated `main` must call after the cyclic code ends.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node.

Command line: Use the option `-functions-called-after-loop`. See “Command-Line Information” on page 1-217.

Settings

No Default

Click  to add a field. Enter function name.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

Tips

- If you specify a function for the option `Initialization functions (-functions-called-before-loop)`, you cannot specify it for this option.

Command-Line Information

Parameter: `-functions-called-after-loop`
No Default
Value: *`function1`*`[,`*`function2`*`[,...]]`
Example (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-main-generator -functions-called-after-loop myfunc`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-main-generator -functions-called-after-loop myfunc`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-main-generator -functions-called-after-loop myfunc`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -functions-called-after-loop myfunc`

See Also

`Initialization functions (-functions-called-before-loop)` | `Step functions (-functions-called-in-loop)` | `Verify model generated code (-main-generator)`

Topics

“Configure Advanced Polyspace Options in Simulink”
“How Polyspace Analysis of Generated Code Works”

Parameters (-variables-written-before-loop)

Specify variables that the generated `main` must initialize before the cyclic code loop

Description

Use this option only for code generated from MATLAB code or Simulink models.

Specify variables that the generated `main` must initialize before the cyclic code loop begins. Before the loop begins, Polyspace considers these variables to have any value allowed by their type.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node.

Command line: Use the option `-variables-written-before-loop`. See "Command-Line Information" on page 1-219.

Settings

Default: `none`

`none`

The generated `main` does not initialize variables.

`all`

The generated `main` initializes all variables except those declared with keyword `const`.

`custom`

The generated `main` only initializes variables that you specify. Click  to add a field. Enter variable name. For C++ class members, use the syntax `className::variableName`.

Command-Line Information

Parameter: `-variables-written-before-loop`
Value: `none` | `all` | `custom=`*`variable1`*`[,`*`variable2`*`[,...]]`
Default: `none`
Example (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-main-generator -variables-written-before-loop all`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-main-generator -variables-written-before-loop all`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-main-generator -variables-written-before-loop all`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -variables-written-before-loop all`

See Also

`Inputs (-variables-written-in-loop)` | `Verify model generated code (-main-generator)`

Topics
“Configure Advanced Polyspace Options in Simulink”
“How Polyspace Analysis of Generated Code Works”

Inputs (-variables-written-in-loop)

Specify variables that the generated `main` must initialize in the cyclic code loop

Description

Use this option only for code generated from MATLAB code or Simulink models.

Specify variables that the generated `main` must initialize at the beginning of every iteration of the cyclic code loop. At the beginning of every loop iteration, Polyspace considers these variables to have any value allowed by their type.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node.

Command line: Use the option `-variables-written-in-loop`. See “Command-Line Information” on page 1-221.

Settings

Default: `none`

`none`

The generated `main` does not initialize variables.

`all`

The generated `main` initializes all variables except those declared with keyword `const`.

`custom`

The generated `main` only initializes variables that you specify. Click  to add a field. Enter variable name. For C++ class members, use the syntax `className::variableName`.

Command-Line Information

Parameter: `-variables-written-in-loop`
Value: `none` | `all` | `custom=`*`variable1`*`[,`*`variable2`*`[,...]]`
Default: `none`
Example (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-main-generator -variables-written-in-loop all`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-main-generator -variables-written-in-loop all`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-main-generator -variables-written-in-loop all`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -variables-written-in-loop all`

See Also

`Parameters (-variables-written-before-loop)` | `Verify model generated code (-main-generator)`

Topics
“Configure Advanced Polyspace Options in Simulink”
“How Polyspace Analysis of Generated Code Works”

Verify module or library (-main-generator)

Generate a main function if source files are modules or libraries that do not contain a main

Description

This option affects a Code Prover analysis only.

Specify that Polyspace must generate a main function if it does not find one in the source files.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node.

Command line: Use the option -main-generator. See “Command-Line Information” on page 1-224.

For the analogous option for model generated code, see Verify model generated code (-main-generator).

Why Use This Option

Use this option if you are verifying a module or library. A Code Prover analysis requires a main function. When verifying a module or library, your code might not have a main.

When you use this option, Code Prover generates a main function if one does not exist. If a main exists, the analysis uses the existing main.

Settings

On (default)

Polyspace generates a main function if it does not find one in the source files. The generated main:

1. Initializes variables specified by Variables to initialize (-main-generator-writes-variables).
2. Before calling other functions, calls the functions specified by Initialization functions (-functions-called-before-main).
3. In all possible orders, calls the functions specified by Functions to call (-main-generator-calls).
4. (C++ only) Calls class methods specified by Class (-class-analyzer) and Functions to call within the specified classes (-class-analyzer-calls).

If you do not specify the function and variable options above, the generated main:

- Initializes all global variables except those declared with keywords const and static.
- In all possible orders, calls all functions that are not called anywhere in the source files. Polyspace considers that global variables can be written between two consecutive function calls. Therefore, in each called function, global variables initially have the full range of values allowed by their type.

Off

Polyspace stops if a `main` function is not present in the source files.

Tips

- If a `main` function is present in your source files, the verification uses that `main` function, irrespective of whether you enable or disable this option.

  The option is relevant only if a `main` function is not present in your source files.
- If you use the option `Verify whole application` (default on the command line), your code must contain a `main` function. Otherwise you see the error:

  ```
  Error: required main procedure not found
  ```

  If your code does not contain a `main` function, use this option to generate a `main` function.
- If you specify multitasking options, the verification ignores your specifications for `main` generation. Instead, the verification introduces an empty `main` function.

  For more information on the multitasking options, see “Configuring Polyspace Multitasking Analysis Manually”.

Command-Line Information

Parameter: `-main-generator`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-main-generator`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-main-generator`

See Also

`Class (-class-analyzer)` | `Functions to call (-main-generator-calls)` | `Functions to call within the specified classes (-class-analyzer-calls)` | `Initialization functions (-functions-called-before-main)` | `Variables to initialize (-main-generator-writes-variables)` | `Verify whole application`

Topics
“Specify Polyspace Analysis Options”
“Verify C Application Without main Function” (Polyspace Code Prover)

Consider volatile qualifier on fields (-consider-volatile-qualifier-on-fields)

Assume that volatile qualified structure fields can have all possible values at any point in code

Description

This option affects a Code Prover analysis only.

Specify that the verification must take into account the volatile qualifier on fields of a structure.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Verification Assumptions** node.

Command line: Use the option -consider-volatile-qualifier-on-fields. See "Command-Line Information" on page 1-227.

Why Use This Option

The volatile qualifier on a variable indicates that the variable value can change between successive operations even if you do not explicitly change it in your code. For instance, if var is a volatile variable, the consecutive operations res = var; res =var; can result in two different values of var being read into res.

Use this option so that the verification emulates the volatile qualifier for structure fields. If you select this option, the software assumes that a volatile structure field has a full range of values at any point in the code. The range is determined only by the data type of the structure field.

Settings

☑ On

The verification considers the volatile qualifier on fields of a structure.

In the following example, the verification considers that the field val1 can have all values allowed for the int type at any point in the code.

```
struct myStruct {
   volatile int val1;
   int val2;
};
```

Even if you write a specific value to val1 and read the variable in the next operation, the variable read results in any possible value.

```
struct myStruct myStructInstance;
myStructInstance.val1 = 1;
assert (myStructInstance.val1 == 1); // Assertion can fail
```

Off (default)

The verification ignores the `volatile` qualifier on fields of a structure.

In the following example, the verification ignores the qualifier on field `val1`.

```
struct myStruct {
   volatile int val1;
   int val2;
};
```

If you write a specific value to `val1` and read the variable in the next operation, the variable read results in that specific value.

```
struct myStruct myStructInstance;
myStructInstance.val1 = 1;
assert (myStructInstance.val1 == 1); // Assertion passes
```

Tips

- If your volatile fields do not represent values read from hardware and you do not expect their values to change between successive operations, disable this option. You are using the `volatile` qualifier for some other reason and the verification does not need to consider full range for the field values.
- If you enable this option, the number of red, gray, and green checks in your code can decrease. The number of orange checks can increase.

  In the following example, a red or green check changes to orange or a gray check goes away when the option is used. Considering the `volatile` qualifier changes the check color. These examples use the following structure definition:

```
struct myStruct {
   volatile int field1;
   int field2;
};
```

Color Without Option	Result Without Option	Result With Option

Green	`void main(){`
	`  struct myStruct structVal;`
	`  structVal.field1 = 1;`
	`  assert(structVal.field1 == 1);`
	`}`
		`void main(){`
		`  struct myStruct structVal;`
		`  structVal.field1 = 1;`
		`  assert(structVal.field1 ==1);`
		`}`

Red	`void main(){`
	`  struct myStruct structVal;`
	`  structVal.field1 = 1;`
	`  assert(structVal.field1 != 1);`
	`}`
		`void main(){`
		`  struct myStruct structVal;`
		`  structVal.field1 = 1;`
		`  assert(structVal.field1 !=1);`
		`}`

Color Without Option	Result Without Option	Result With Option

Gray	`void main(){`
	`  struct myStruct structVal;`
	`  structVal.field1 = 1;`
	`  if (structVal.field1 != 1)`
	`  {`
	`  /* Perform operation */`
	`  }`
	`}`
		`void main(){`
		`  struct myStruct structVal;`
		`  structVal.field1 = 1;`
		`  if (structVal.field1 != 1)`
		`  {`
		`  /* Perform operation */`
		`  }`
		`}`

- In C++ code, the option also applies to class members.

Command-Line Information

Parameter: `-consider-volatile-qualifier-on-fields`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-consider-volatile-qualifier-on-fields`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-consider-volatile-qualifier-on-fields`

See Also

Topics

“Specify Polyspace Analysis Options”

Introduced in R2016b

Float rounding mode (`-float-rounding-mode`)

Specify rounding modes to consider when determining the results of floating point arithmetic

Description

This option affects a Code Prover analysis only.

Specify the rounding modes to consider when determining the results of floating-point arithmetic.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Verification Assumptions** node.

Command line: Use the option `-float-rounding-mode`. See “Command-Line Information” on page 1-230.

Why Use This Option

The default verification uses the round-to-nearest mode.

Use the rounding mode `all` if your code contains routines such as `fesetround` to specify a rounding mode other than round-to-nearest. Although the verification ignores the `fesetround` specification, it considers all rounding modes including the rounding mode that you specified. Alternatively, for targets that can use extended precision (for instance, using the flag `-mfpmath=387`), use the rounding mode `all`. However, for your Polyspace analysis results to agree with run-time behavior, you must prevent use of extended precision through a flag such as `-ffloat-store`.

Otherwise, continue to use the default rounding mode `to-nearest`. Because all rounding modes are considered when you specify `all`, you can have many orange **Overflow** checks resulting from overapproximation.

Settings

Default: `to-nearest`

`to-nearest`

The verification assumes the round-to-nearest mode.

`all`

The verification assumes all rounding modes for each operation involving floating-point variables. The following rounding modes are considered: round-to-nearest, round-towards-zero, round-towards-positive-infinity, and round-towards-negative-infinity.

Tips

- The Polyspace analysis uses floating-point arithmetic that conforms to the IEEE® 754 standard. For instance, the arithmetic uses floating point instructions present in the SSE instruction set. The GNU C flag `-mfpmath=sse` enforces use of this instruction set. If you use the GNU C compiler

with this flag to compile your code, your Polyspace analysis results agree with your run-time behavior.

However, if your code uses extended precision, for instance using the GNU C flag `-mfpmath=387`, your Polyspace analysis results might not agree with your run-time behavior in some corner cases. See some examples of these corner cases in `codeprover_limitations.pdf` in *`polyspaceroot`*`\polyspace\verifier\code_prover_desktop`. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

To prevent use of extended precision, on targets without SSE support, you can use a flag such as `-ffloat-store`. For your Polyspace analysis, use `all` for rounding mode to account for double rounding.

- The **Overflow** check uses the rounding modes that you specify. For instance, the following table shows the difference in the result of the check when you change your rounding modes.

Rounding mode: `to-nearest`	Rounding mode: `all`
If results of floating-point operations are rounded to nearest values: • In the first addition operation, `eps1` is just large enough that the value nearest to `FLT_MAX + eps1` is greater than `FLT_MAX`. The **Overflow** check is red. • In the second addition operation, `eps2` is just small enough that the value nearest to `FLT_MAX + eps2` is `FLT_MAX`. The **Overflow** check is green.	Besides to-nearest mode, the **Overflow** check also considers other rounding modes. • In the first addition operation, in to-nearest mode, the value nearest to `FLT_MAX + eps1` is greater than `FLT_MAX`, so the addition overflows. But if rounded towards negative infinity, the result is `FLT_MAX`, so the addition does not overflow. Combining these two rounding modes, the **Overflow** check is orange. • In the second addition operation, in to-nearest mode, the value nearest to `FLT_MAX + eps2` is `FLT_MAX`, so the addition does not overflow. But if rounded towards positive infinity, the result is greater than `FLT_MAX`, so the addition overflows. Combining these two rounding modes, the **Overflow** check is orange.

`#include <float.h>`
`#define eps1 0x1p103`
`#define eps2 0x0.FFFFFFp103`
 	
`float func(int ch) {`
`  float left_op = FLT_MAX;`
`  float right_op_1 = eps1, \`
`right_op_2 = eps2;`
`  switch(ch) {`
`  case 1:`
`  return (left_op +\`
`right_op_1);`
`  case 2:`
`  return (left_op +\`
`right_op_2);`
`  default:`
`  return 0;`
`  }`
`}`
	`#include <float.h>`
	`#define eps1 0x1p103`
	`#define eps2 0x0.FFFFFFp103`
	
	`float func(int ch) {`
	`  float left_op = FLT_MAX;`
	`  float right_op_1 = eps1, \`
	`right_op_2 = eps2;`
	`  switch(ch) {`
	`  case 1:`
	`  return (left_op +\`
	`right_op_1);`
	`  case 2:`
	`  return (left_op +\`
	`right_op_2);`
	`  default:`
	`  return 0;`
	`  }`
	`}`

If you set the rounding mode to `all` and obtain an orange **Overflow** check, to determine how the overflow can occur, consider all rounding modes.

Command-Line Information

Parameter: `-float-rounding-mode`
Value: `to-nearest` | `all`
Default: `to-nearest`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-float-rounding-mode all`

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -float-rounding-mode all`

See Also

`Overflow`

Topics

“Specify Polyspace Analysis Options”

Introduced in R2016a

Respect types in fields (`-respect-types-in-fields`)

Do not cast nonpointer fields of a structure to pointers

Description

This option affects a Code Prover analysis only.

Specify that structure fields not declared initially as pointers will not be cast to pointers later.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Verification Assumptions** node.

Command line: Use the option `-respect-types-in-fields`. See “Command-Line Information” on page 1-233.

Why Use This Option

Use this option to identify and forbid casts from nonpointer structure fields to pointers.

Settings

☑ On

The verification assumes that structure fields not declared initially as pointers will not be cast to pointers later.

Code with option off	Code with option on

`struct {`
`  unsigned int x1;`
`  unsigned int x2;`
`} S;`
 	
`void funct(void) {`
`  int var, *tmp;`
`  S.x1 = &var;`
`  tmp = (int*)S.x1;`
`  *tmp = 1;`
`  assert(var==1);`
`}`
 	
In this example, the fields of `S` are declared as integers but `S.x1` is cast to a pointer. With the option turned off, Polyspace allows the cast.
	`struct {`
	`  unsigned int x1;`
	`  unsigned int x2;`
	`} S;`
	
	`void funct(void) {`
	`  int var, *tmp;`
	`  S.x1 = &var;`
	`  tmp = (int*)S.x1;`
	`  *tmp = 1;`
	`  assert(var==1);`
	`}`
	
	In this example, the fields of `S` are declared as integers but `S.x1` is cast to a pointer. With the option turned on, Polyspace ignores the cast. Therefore, it ignores the initialization of `var` through the pointer `(int*)S.x1` and produces a red **Non-initialized local variable** error when `var` is read.

Off (default)

The verification assumes that structure fields can be cast to pointers even when they are not declared as pointers.

Command-Line Information

Parameter: `-respect-types-in-fields`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources file_name -respect-types-in-fields`
Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -respect-types-in-fields`

See Also

`Non-initialized local variable` | `Respect types in global variables (-respect-types-in-globals)`

Topics

“Specify Polyspace Analysis Options”

Respect types in global variables (`-respect-types-in-globals`)

Do not cast nonpointer global variables to pointers

Description

This option affects a Code Prover analysis only.

Specify that global variables not declared initially as pointers will not be cast to pointers later.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Verification Assumptions** node.

Command line: Use the option `-respect-types-in-globals`. See “Command-Line Information” on page 1-235.

Why Use This Option

Use this option to identify and forbid casts from nonpointer global variables to pointers.

Settings

☑ On

The verification assumes that global variables not declared initially as pointers will not be cast to pointers later.

☐ Off (default)

The verification assumes that global variables can be cast to pointers even when they are not declared as pointers.

Tips

If you select this option, the number of checks in your code can change. You can use this option and the change in results to identify cases where you cast nonpointer variables to pointers.

For instance, in the following example, when you select the option, the results have one less orange check and one more red check.

Code with option off	Code with option on

`int global;`
`void main(void) {`
`  int local;`
`  global = (int)&local;`
`  *(int*)global = 5;`
`  assert(local==5);`
`}`
 	
In this example, `global` is declared as an `int` variable but cast to a pointer. With the option turned off, Polyspace allows the cast.
	`int global;`
	`void main(void) {`
	`  int local;`
	`  global = (int)&local;`
	`  *(int*)global = 5;`
	`  assert(local==5);`
	`}`
	
	In this example, `global` is declared as an `int` variable but cast to a pointer. With the option turned on, Polyspace ignores the cast. Therefore, it ignores the initialization of `local` through the pointer `(int*)global` and produces a red **Non-initialized local variable** error when `local` is read.

Command-Line Information

Parameter: `-respect-types-in-globals`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-respect-types-in-globals`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-respect-types-in-globals`

See Also

`Non-initialized local variable` | `Respect types in fields (-respect-types-in-fields)`

Topics
“Specify Polyspace Analysis Options”

Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)

Specify that environment pointers can be unsafe to dereference unless constrained otherwise

Description

This option affects a Code Prover analysis only.

This option is not available for code generated from MATLAB code or Simulink models.

Specify that the verification must consider environment pointers as unsafe unless otherwise constrained. Environment pointers are pointers that can be assigned values outside your code.

Environment pointers include:

- Global or `extern` pointers.
- Pointers returned from stubbed functions.

  A function is stubbed if your code does not contain the function definition or you override a function definition by using the option `Functions to stub (-functions-to-stub)`.
- Pointer parameters of functions whose calls are generated by the software.

  A function call is generated if you verify a module or library and the module or library does not have an explicit call to the function. You can also force a function call to be generated with the option `Functions to call (-main-generator-calls)`.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Verification Assumptions** node.

Command line: Use the option `-stubbed-pointers-are-unsafe`. See “Command-Line Information” on page 1-238.

Why Use This Option

Use this option so that the verification makes more conservative assumptions about pointers from external sources.

If you specify this option, the verification considers that environment pointers can have a `NULL` value. If you read an environment pointer without checking for `NULL`, the **Illegally dereferenced pointer** check shows a potential error in orange. The message associated with the orange check shows the pointer can be `NULL`.

Settings

☑ On

The verification considers that environment pointers can have a `NULL` value.

Off (default)

The verification considers that environment pointers:

- Cannot have a NULL value.
- Points within allowed bounds.

Tips

- Enable this option during the integration phase. In this phase, you provide complete code for verification. Even if an orange check originates from external sources, you are likely to place protections against unsafe pointers from such sources. For instance, if you obtain a pointer from an unknown source, you check the pointer for NULL value.

  Disable this option during the unit testing phase. In this phase, you focus on errors originating from your unit.
- If you are verifying code implementation of AUTOSAR runnables, Code Prover assumes that pointer arguments to runnables and pointers returned from Rte_ functions are not NULL. You cannot use this option to change the assumption. See “Run Polyspace on AUTOSAR Code with Conservative Assumptions” (Polyspace Code Prover).
- If you enable this option, the number of orange checks in your code might increase.

Environment Pointers Safe	Environment Pointers Unsafe

The **Illegally dereferenced pointer** check is green. The verification assumes that env_ptr is not NULL and any dereference is within allowed bounds. The verification assumes that the result of the dereference is full range. For instance, in this case, the return value has the full range of type int.
`int func (int *env_ptr) {`
`  return *env_ptr;`
`}`
	The **Illegally dereferenced pointer** check is orange. The verification assumes that env_ptr can be NULL.
	`int func (int *env_ptr) {`
	`  return *env_ptr;`
	`}`

  If you enable this option, the number of gray checks might decrease.

Environment Pointers Safe	Environment Pointers Unsafe

The verification assumes that env_ptr is not NULL. The if condition is always true and the else block is unreachable.
`#include <stdlib.h>`
`int func (int *env_ptr) {`
`  if(env_ptr!=NULL)`
`  return *env_ptr;`
`  else`
`  return 0;`
`}`
	The verification assumes that env_ptr can be NULL. The if condition is not always true and the else block can be reachable.
	`#include <stdlib.h>`
	`int func (int *env_ptr) {`
	`  if(env_ptr!=NULL)`
	`  return *env_ptr;`
	`  else`
	`  return 0;`
	`}`

- Instead of considering all environment pointers as safe or unsafe, you can individually constrain some of the environment pointers. See the description of **Initialize Pointer** in “External Constraints for Polyspace Analysis” (Polyspace Code Prover).

When you individually constrain a pointer, you first specify an **Init Mode**, and then specify through the **Initialize Pointer** option whether the pointer is `Null`, `Not Null`, or `Maybe Null`. Depending on the **Init Mode**, you can either override the global specification for all environment pointers or not.

- If you set the **Init Mode** of the pointer to `INIT` or `PERMANENT`, your selection for **Initialize Pointer** overrides your specification for this option. For instance, if you specify `Not NULL` for an environment pointer `ptr`, the verification assumes that `ptr` is not NULL even if you specify that environment pointers must be considered unsafe.
- If you set the **Init Mode** to `MAIN GENERATOR`, the verification uses your specification for this option.

  For pointers returned from stubbed functions, the option `MAIN GENERATOR` is not available. If you override the global specification for such a pointer through the **Initialize Pointer** option in constraints, you cannot toggle back to the global specification without changing the **Initialize Pointer** option too.

- If you disable this option, the verification considers that dereferences at all pointer depths are valid.

  For instance, all the dereferences are considered valid in this code:

```
int*** stub(void);

void func2() {
        int ***ptr = stub();
        int **ptr2 = *ptr;
        int *ptr3 = *ptr2;
}
```

Command-Line Information

Parameter: `-stubbed-pointers-are-unsafe`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources file_name -stubbed-pointers-are-unsafe`
Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -stubbed-pointers-are-unsafe`

See Also

`Constraint setup (-data-range-specifications)`

Topics

“Specify Polyspace Analysis Options”
“Specify External Constraints”
“External Constraints for Polyspace Analysis”

Introduced in R2016b

Allow negative operand for left shifts (-allow-negative-operand-in-shift)

Allow left shift operations on a negative number

Description

This option affects a Code Prover analysis only.

Specify that the verification must allow left shift operations on a negative number.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-allow-negative-operand-in-shift`. See "Command-Line Information" on page 1-239.

Why Use This Option

According to the C99 standard (sec 6.5.7), the result of a left shift operation on a negative number is undefined. Following the standard, the verification produces a red check on left shifts of negative numbers.

If your compiler has a well-defined behavior for left shifts of negative numbers, set this option. Note that allowing left shifts of negative numbers can reduce the cross-compiler portability of your code.

Settings

☑ On

The verification allows shift operations on a negative number, for instance, `-2 << 2`.

☐ Off (default)

If a shift operation is performed on a negative number, the verification generates an error.

Command-Line Information

Parameter: `-allow-negative-operand-in-shift`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources file_name -allow-negative-operand-in-shift`
Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -allow-negative-operand-in-shift`

See Also

`Invalid shift operations`

Topics
“Specify Polyspace Analysis Options”

Consider non finite floats (-allow-non-finite-floats)

Enable an analysis mode that incorporates infinities and NaNs

Description

Enable an analysis mode that incorporates infinities and NaNs for floating point operations.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-allow-non-finite-floats`. See “Command-Line Information” on page 1-243.

Why Use This Option

Code Prover

By default, the analysis does not incorporate infinities and NaNs. For instance, the analysis terminates the execution thread where a division by zero occurs and does not consider that the result could be infinite.

If you use functions such as `isinf` or `isnan` and account for infinities and NaNs in your code, set this option. When you set this option and a division by zero occurs for instance, the execution thread continues with infinity as the result of the division.

Set this option alone if you are sure that you have accounted for infinities and NaNs in your code. Using the option alone effectively disables many numerical checks on floating point operations. If you have generally accounted for infinities and NaNs, but you are not sure that you have considered all situations, set these additional options:

- `Infinities (-check-infinite)`: Use `warn-first`.
- `NaNs (-check-nan)`: Use `warn-first`.

Bug Finder

If the analysis flags comparisons using `isinf` or `isnan` as dead code, use this option. By default, a Bug Finder analysis does not incorporate infinities and NaNs.

Settings

☑ On

The analysis allows infinities and NaNs. For instance, in this mode:

- The analysis assumes that floating-point operations can produce results such as infinities and NaNs.

By using options `Infinities (-check-infinite)` and `NaNs (-check-nan)`, you can choose to highlight operations that produce nonfinite results and stop the execution threads where the nonfinite results occur. These options are not available for a Bug Finder analysis.

- The analysis assumes that floating-point variables with unknown values can have any value allowed by their type, including infinite or NaN. Floating-point variables with unknown values include volatile variables and return values of stubbed functions.

☐ Off (default)

The analysis does not allow infinities and NaNs. For instance, in this mode:

- The Code Prover analysis produces a red check on a floating-point operation that produces an infinity or a NaN as the only possible result on all execution paths. The verification produces an orange check on a floating-point operation that can potentially produce an infinity or NaN.
- The Code Prover analysis assumes that floating-point variables with unknown values are full-range but finite.
- The Bug Finder analysis shows comparisons with infinity using `isinf` as dead code.

Tips

- The IEEE 754 Standard allows special quantities such as infinities and `NaN` so that you can handle certain numerical exceptions without aborting the code. Some implementations of the C standard support infinities and `NaN`.
  - If your compiler supports infinities and `NaN`s and you account for them explicitly in your code, use this option so that the verification also allows them.

    For instance, if a division results in infinity, in your code, you specify an alternative action. Therefore, you do not want the verification to highlight division operations that result in infinity.
  - If your compiler supports infinities and `NaN`s but you are not sure if you account for them explicitly in your code, use this option so that the verification incorporates infinities and `NaN`s. Use the options `-check-nan` and `-check-infinite` with argument `warn` so that the verification highlights operations that result in infinities and `NaN`s, but does not stop the execution thread. These options are not available for a Bug Finder analysis.
- If you run a Code Prover analysis and use this option, checkers for overflow, division by zero and other numerical run-time errors are disabled. See “Numerical Checks” (Polyspace Code Prover).

  If you run a Bug Finder analysis and use this option:
  - The checkers for overflow and division by zero are disabled. See “Numerical Defects”.
  - The checker `Floating point comparison with equality operators` can show false positives.
- If you select this option, the number and type of Code Prover checks in your code can change.

  For instance, in the following example, when you select the option, the results have one less red check and three more green checks.

Infinities and NaNs Not Allowed	Infinities and NaNs Allowed

Code Prover produces a **Division by zero** error and stops verification.
`double func(void) {`
`  double x=1.0/0.0;`
`  double y=1.0/x;`
`  double z=x-x;`
`  return z;`
`}`
	If you select this option, Code Prover does not check for a **Division by zero** error.
	`double func(void) {`
	`  double x=1.0/0.0;`
	`  double y=1.0/x;`
	`  double z=x-x;`
	`  return z;`
	`}`
	The analysis assumes that dividing by zero results in:
	• Value of `x` equal to `Inf`
	• Value of `y` equal to 0.0
	• Value of `z` equal to `NaN`
	In your analysis results in the Polyspace user interface, if you place your cursor on `y` and `z`, you can see the nonfinite values `Inf` and `NaN` respectively in the tooltip.

- You cannot run the Automatic Orange Tester in Code Prover if you incorporate non-finites in your analysis.

Command-Line Information

Parameter: `-allow-non-finite-floats`
Default: Off
Example (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-allow-non-finite-floats`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-allow-non-finite-floats`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-allow-non-finite-floats`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-allow-non-finite-floats`

See Also

“Numerical Defects” | “Numerical Checks” (Polyspace Code Prover) | `Infinities (-check-infinite)` | `NaNs (-check-nan)`

Topics
“Specify Polyspace Analysis Options”

Introduced in R2016a

Infinities (-check-infinite)

Specify how to handle floating-point operations that result in infinity

Description

This option affects a Code Prover analysis only.

Specify how the analysis must handle floating-point operations that result in infinities.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node. See “Dependencies” on page 1-245 for other options you must also enable.

Command line: Use the option `-check-infinite`. See “Command-Line Information” on page 1-245.

Why Use This Option

Use this option to enable detection of floating-point operations that result in infinities.

If you specify that the analysis must consider nonfinite floats, by default, the analysis does not flag these operations. Use this option to detect these operations while still incorporating nonfinite floats.

Settings

Default: `allow`

`allow`

The verification does not produce a check on the operation.

For instance, in the following code, there is no **Overflow** check.

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

`warn-first`

The verification produces a check on the operation. The check determines if the result of the operation is infinite when the operands themselves are not infinite. The verification does not terminate the execution thread that produces infinity.

If the verification detects an operation that produces infinity as the only possible result on all execution paths and the operands themselves are never infinite, the check is red. If the operation can potentially result in infinity, the check is orange.

For instance, in the following code, there is a nonblocking **Overflow** check for infinity.

```
double func(void) {
    double x=1.0/0.0;
```

```
    return x;
}
```

Even though the **Overflow** check on the / operation is red, the verification continues. For instance, a green **Non-initialized local variable** check appears on x in the return statement.

`forbid`

The verification produces a check on the operation and terminates the execution thread that produces infinity.

If the check is red, the verification does not continue for the remaining code in the same scope as the check. If the check is orange, the verification continues but removes from consideration the variable values that produced infinity.

For instance, in the following code, there is a blocking **Overflow** check for infinity.

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

The verification stops because the **Overflow** check on the / operation is red. For instance, a **Non-initialized local variable** check does not appear on x in the return statement.

Dependencies

To use this option, you must enable the verification mode that incorporates infinities and NaNs. See Consider non finite floats (-allow-non-finite-floats).

Command-Line Information

Parameter: -check-infinite
Value: allow | warn-first | forbid
Default: allow
Example (Code Prover): polyspace-code-prover -sources *file_name* -check-infinite forbid
Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -check-infinite forbid

See Also

Polyspace Analysis Options

Consider non finite floats (-allow-non-finite-floats) | NaNs (-check-nan)

Polyspace Results

Overflow

Topics

“Specify Polyspace Analysis Options”

Introduced in R2016a

Check that global variables are initialized after warm reboot (`-check-globals-init`)

Check that global variables are assigned values in designed initialization code

Description

This option affects a Code Prover analysis only.

Specify that Polyspace must check whether all non-const global variables (and local static variables) are explicitly initialized at declaration or within a section of code marked as initialization code.

To indicate the end of initialization code, you enter the line

```
#pragma polyspace_end_of_init
```

in the `main` function (only once). The initialization code starts from the beginning of `main` and continues up to this pragma.

Since compilers ignore unrecognized pragmas, the presence of this pragma does not affect program execution.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-check-globals-init`. See “Command-Line Information” on page 1-249.

Why Use This Option

In a warm reboot, to save time, the bss segment of a program, which might hold variable values from a previous state, is not loaded. Instead, the program is supposed to explicitly initialize all non-const variables without default values before execution. You can use this option to delimit the initialization code and verify that all non-const global variables are indeed initialized in a warm reboot.

For instance, in this simple example, the global variable `aVar` is initialized in the initialization code section but the variable `anotherVar` is not.

```
int aVar;
const int aConst = -1;
int anotherVar;

int main() {
      aVar = aConst;
#pragma polyspace_end_of_init
      return 0;
}
```

Settings

☑ On

Polyspace checks whether all global variables are initialized in the designated initialization code. The initialization code starts from the beginning from `main` and continues up to the pragma `polyspace_end_of_init`.

The results are reported using the check `Global variable not assigned a value in initialization code`.

○ Off (default)

Polyspace does not check for initialization of global variables in a designated code section.

However, the verification continues to check if a variable is initialized at the time of use. The results are reported using the check `Non-initialized variable`.

Dependencies

You can use this option and designate a section of code as initialization code only if:

- Your program contains a `main` function and you use the option `Verify whole application` (implicitly set by default at command line).
- You set `Source code language (-lang)` to `C`.

Note that the pragma must appear only once in the `main` function. The pragma can appear before or after variable declarations but must appear after type definitions (`typedef`-s).

You cannot use this option with the following options:

- `Disable checks for non-initialization (-disable-initialization-checks)`
- `Verify files independently (-unit-by-unit)`
- `Show global variable sharing and usage only (-shared-variables-mode)`

Tips

- You can use this option along with the option `Verify initialization section of code only (-init-only-mode)` to check the initialization code before checking the remaining program.

  This approach has the following benefits compared to checking the entire code in one run:

  - Run-time errors in the initialization code can invalidate analysis of the remaining code. You can run a comparatively quicker check on the initialization code before checking the remaining program.
  - You can review results of the checker `Global variable not assigned a value in initialization code` relatively easily.

    Consider this example. There is an orange check on `var` because `var` might remain uninitialized when the `if` and `else if` statements are skipped.

    ```
    int var;
    ```

```
int checkSomething(void);
int checkSomethingElse(void);

int main() {
    int local_var;
    if(checkSomething())
    {
        var=0;
    }
    else if(checkSomethingElse()) {
        var=1;
    }
    #pragma polyspace_end_of_init
    var=2;
    local_var = var;
    return 0;
}
```

To review this check and understand when `x` might be non-initialized, you have to browse through all instances of `x` on the **Variable Access** pane. If you check the initialization code alone, only the code in bold gets checked and you have to browse through only the instances in the initialization code.

- The check is only as good as your placement of the pragma `polyspace_end_of_init`. For instance:

  - Place the pragma only after initialization code ends.

    Otherwise, a variable might appear falsely uninitialized.

  - Try to place the pragma directly in the `main` function, that is, outside a block. If you place the pragma in a block, the check considers only those paths that end in the block.

    All paths that end in the block might have a variable initialized but paths that skip the block might let the variable go uninitialized. If you do place the pragma in a block, make sure that it is okay if a variable stays uninitialized outside the block.

    For instance, in this example, the variable `var` is initialized on all paths that end at the location of the pragma. The check is green despite the fact that the `if` block might be skipped, letting the variable go uninitialized.

    ```
    int var;

    int func();

    int main() {
        int err = func();
        if(err) {
            var = 0;
     #pragma polyspace_end_of_init
        }
        int a = var;
        return 0;
    }
    ```

    The issue is detected by the checker if you place the pragma after the `if` block ends.

- Do not place the pragma in a loop.

  If you place the pragma in a loop, you can see results that are difficult to interpret. For instance, in this example, both `aVar` and `anotherVar` are initialized in one iteration of the loop. However, the pragma only considers the first iteration of the loop when it shows a green check for initialization. If a variable is initialized on a later iteration, the check is orange.

  ```
  int aVar;
  int anotherVar;

  void main() {
      for(int i=0; i<=1; i++) {
          if(i == 0)
              aVar = 0;
          else
              anotherVar = 0;
          #pragma polyspace_end_of_init
      }
  }
  ```

  The check is red if you verify initialization code alone and do not initialize a variable in the first loop iteration. To avoid these incorrect red or orange checks, do not place the pragma in a loop.
- To determine the initialization of a structure, a regular Code Prover analysis only considers fields that are used.

  If you check initialization code only using the option `Verify initialization section of code only (-init-only-mode)`, the analysis covers only a portion of the code and cannot determine if a variable is used beyond this portion. Therefore, the checks for initialization consider all structure fields, whether used or not.

Command-Line Information

Parameter: `-check-globals-init`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-check-globals-init`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-check-globals-init`

See Also

`Verify initialization section of code only (-init-only-mode)`

Topics

“Specify Polyspace Analysis Options”

Introduced in R2020a

NaNs (-check-nan)

Specify how to handle floating-point operations that result in NaN

Description

This option affects a Code Prover analysis only.

Specify how the analysis must handle floating-point operations that result in NaN.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node. See “Dependencies” on page 1-251 for other options you must also enable.

Command line: Use the option `-check-nan`. See “Command-Line Information” on page 1-251.

Why Use This Option

Use this option to enable detection of floating-point operations that result in NaN-s.

If you specify that the analysis must consider nonfinite floats, by default, the analysis does not flag these operations. Use this option to detect these operations while still incorporating nonfinite floats.

Settings

Default: `allow`

`allow`

The verification does not produce a check on the operation.

For instance, in the following code, there is no **Invalid operation on floats** check.

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
    return y;
}
```

`warn-first`

The verification produces a check on the operation. The check determines if the result of the operation is NaN when the operands themselves are not NaN. For instance, the check flags the operation `val1 + val2` only if the result can be NaN when *both* `val1` and `val2` are not NaN. The verification does not terminate the execution thread that produces NaN.

If the verification detects an operation that produces NaN as the only possible result on all execution paths and the operands themselves are never NaN, the check is red. If the operation can potentially result in NaN, the check is orange.

For instance, in the following code, there is a nonblocking **Invalid operation on floats** check for NaN.

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
    return y;
}
```

Even though the **Invalid operation on floats** check on the `-` operation is red, the verification continues. For instance, a green **Non-initialized local variable** check appears on `y` in the `return` statement.

`forbid`

The verification produces a check on the operation and terminates the execution thread that produces NaN.

If the check is red, the verification does not continue for the remaining code in the same scope as the check. If the check is orange, the verification continues but removes from consideration the variable values that produced a NaN.

For instance, in the following code, there is a blocking **Invalid operation on floats** check for NaN.

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
    return y;
}
```

The verification stops because the **Invalid operation on floats** check on the `-` operation is red. For instance, a **Non-initialized local variable** check does not appear on `y` in the `return` statement.

The **Invalid operation on floats** check for NaN also appears on the `/` operation and is green.

Dependencies

To use this option, you must enable the verification mode that incorporates infinities and NaNs. See `Consider non finite floats (-allow-non-finite-floats)`.

Command-Line Information

Parameter: `-check-nan`
Value: `allow` | `warn-first` | `forbid`
Default: `allow`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-check-nan forbid`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-check-nan forbid`

See Also

Polyspace Analysis Options
`Consider non finite floats (-allow-non-finite-floats)` | `Infinities (-check-infinite)`

Polyspace Results
`Invalid operation on floats`

Topics
“Specify Polyspace Analysis Options”

Introduced in R2016a

Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)

Allow arithmetic on pointer to a structure field so that it points to another field

Description

This option affects a Code Prover analysis only.

Specify that a pointer assigned to a structure field can point outside its bounds as long as it points within the structure.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node. See “Dependency” on page 1-254 for other options you must also enable.

Command line: Use the option `-allow-ptr-arith-on-struct`. See “Command-Line Information” on page 1-254.

Why Use This Option

Use this option to relax the check for illegally dereferenced pointers. Once you assign a pointer to a structure field, you can perform pointer arithmetic and use the result to access another structure field.

Settings

☑ On

A pointer assigned to a structure field can point outside the bounds imposed by the field as long as it points within the structure. For instance, in the following code, unless you use this option, the verification will produce a red `Illegally dereferenced pointer` check:

```
void main(void) {
struct S {char a; char b; int c;} x;
char *ptr = &x.b;
ptr ++;
*ptr = 1; // Red on the dereference, because ptr points outside x.b
}
```

☐ Off (default)

A pointer assigned to a structure field can point only within the bounds imposed by the field.

Tips

- The verification does not allow a pointer with negative offset values. This behavior occurs irrespective of whether you choose the option **Enable pointer arithmetic across fields**.
- Using this option can slightly increase the number of orange checks. The option relaxes the constraint that a pointer to a structure field cannot point to other fields of the structure. In

exchange for relaxing this constraint, the verification loses precision on the boundary of fields within a structure and treats the structure as a whole. Pointer dereferences that were previously green can now turn orange.

Use this option if you follow a policy of reviewing red checks only and you need to work around red checks from pointer arithmetic within a structure.

- Before using this option, consider the costs of using pointer arithmetic across different fields of a structure.

  Unlike an array, members of a structure can have different data types. For efficient storage, structures use padding to accommodate this difference. When you increment a pointer pointing to a structure member, you might not point to the next member. When you dereference this pointer, you cannot rely on what you are reading or writing to.

Dependency

This option is available only if you set `Source code language (-lang)` to `C`.

Command-Line Information

Parameter: `-allow-ptr-arith-on-struct`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources file_name -allow-ptr-arith-on-struct`
Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -allow-ptr-arith-on-struct`

See Also

`Allow incomplete or partial allocation of structures (-size-in-bytes)` | `Illegally dereferenced pointer`

Topics
“Specify Polyspace Analysis Options”

Detect stack pointer dereference outside scope (-detect-pointer-escape)

Find cases where a function returns a pointer to one of its local variables

Description

This option affects a Code Prover analysis only.

Specify that the verification must detect cases where you access a variable outside its scope via pointers. Such an access can happen, for example, when a function returns a pointer to a local variable and you dereference the pointer outside the function. The dereference causes undefined behavior because the local variable that the pointer points to does not live outside the function.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-detect-pointer-escape`. See “Command-Line Information” on page 1-256.

Why Use This Option

Use this option to enable detection of pointer escape.

Settings

☑ On

The **Illegally dereferenced pointer** check performs an additional task, besides its usual specifications. When you dereference a pointer, the check also determines if you are accessing a variable outside its scope through the pointer. The check is:

- Red, if all the variables that the pointer points to are accessed outside their scope.

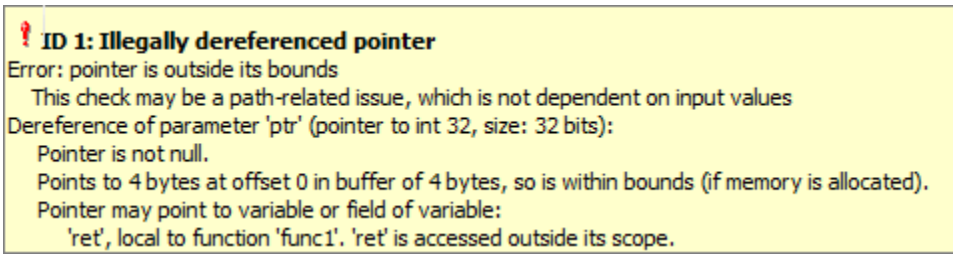
  For instance, you dereference a pointer `ptr` in a function `func` that is called twice in your code. In both calls, when you perform the dereference `*ptr`, `ptr` is pointing to variables outside their scope. Therefore, the **Illegally dereferenced pointer** check is red.
- Orange, if only some of the variables that the pointer points to are accessed outside their scope.
- Green, if none of the variables that the pointer points to are accessed outside their scope, and other requirements of the check are also satisfied.

In the following code, if you enable this option, Polyspace Code Prover produces a red **Illegally dereferenced pointer** check on `*ptr`. Otherwise, the **Illegally dereferenced pointer** check on `*ptr` is green.

```
void func2(int *ptr) {
    *ptr = 0;
}
```

```
int* func1(void) {
    int ret = 0;
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

The **Result Details** pane displays a message indicating that `ret` is accessed outside its scope.



☐ Off (default)

When you dereference a pointer, the **Illegally dereferenced pointer** check does not check for whether you are accessing a variable outside its scope. The check is green even if the pointer dereference is outside the variable scope, as long as it satisfies these requirements:

- The pointer is not NULL.
- The pointer points within the memory buffer.

Tips

The detection of stack pointer deference outside scope does not apply to certain types of pointers. For specific limitations, see “Limitations of Polyspace Verification” (Polyspace Code Prover).

Command-Line Information

Parameter: `-detect-pointer-escape`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-detect-pointer-escape`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-detect-pointer-escape`

See Also

`Illegally dereferenced pointer`

Topics
“Specify Polyspace Analysis Options”

Introduced in R2015a

Disable checks for non-initialization (-disable-initialization-checks)

Disable checks for non-initialized variables and pointers

Description

This option affects a Code Prover analysis only.

Specify that Polyspace Code Prover must not check for non-initialization in your code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-disable-initialization-checks`. See “Command-Line Information” on page 1-258.

Why Use This Option

Use this option if you do not want to detect instances of non-initialized variables.

Settings

☑ On

Polyspace Code Prover does not perform the following checks:

- `Non-initialized local variable`: Local variable is not initialized before being read.
- `Non-initialized variable`: Variable other than local variable is not initialized before being read.
- `Non-initialized pointer`: Pointer is not initialized before being read.
- `Return value not initialized`: C function does not return value when expected.

Polyspace assumes that, at declaration:

- Variables have full-range of values allowed by their type.
- Pointers can be `NULL`-valued or point to a memory block at an unknown offset.

☐ Off (default)

Polyspace Code Prover checks for non-initialization in your code. The software displays red checks if, for instance, a variable is not initialized and orange checks if a variable is initialized only on some execution paths.

Tips

- If you select this option, the software does not report most violations of MISRA C:2004 rule 9.1, and `MISRA C:2012 Rule 9.1`.

- If you select this option, the number and type of orange checks in your code can change.

  For instance, the following table shows an additional orange check with the option enabled.

Checks for Non-initialization Enabled	Checks for Non-initialization Disabled

`void func(int flag) {`
`  int var1,var2;`
`  if( flag==0) {`
`  var1=var2;`
`  }`
`  else {`
`  var1=0;`
`  }`
`  var2=var1 + 1;`
`}`
 	
In this example, the software produces:
• A red **Non-initialized local variable** check on `var2` in the `if` branch. The verification continues as if only the `else` branch of the `if` statement exists.
• A green **Non-initialized local variable** check on `var1` in the last statement. `var1` has the assigned value 0.
• A green **Overflow** check on the + operation.
	`void func(int flag) {`
	`  int var1,var2;`
	`  if( flag==0) {`
	`  var1=var2;`
	`  }`
	`  else {`
	`  var1=0;`
	`  }`
	`  var2=var1 + 1;`
	`}`
	
	In this example, the software:
	• Does not produce **Non-initialized local variable** checks. At initialization, the software assumes that `var2` has full range of `int` values. Following the `if` statement, because the software considers both `if` branches, it assumes that `var1` also has full range of `int` values.
	• Produces an orange **Overflow** check on the + operation. For instance, if `var1` has the maximum `int` value, adding 1 to it can cause an overflow.

Command-Line Information

Parameter: `-disable-initialization-checks`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-disable-initialization-checks`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-disable-initialization-checks`

See Also

Topics
“Specify Polyspace Analysis Options”

Permissive function pointer calls (-permissive-function-pointer)

Allow type mismatch between function pointers and the functions they point to

Description

This option affects a Code Prover analysis only.

Specify that the verification must allow function pointer calls where the type of the function pointer does not match the type of the function.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node. See “Dependency” on page 1-261 for other options you must also enable.

Command line: Use the option `-permissive-function-pointer`. See “Command-Line Information” on page 1-261.

Why Use This Option

By default, Code Prover does not recognize calls through function pointers when a type mismatch occurs. Fix the type mismatch whenever possible.

Use this option if:

- You cannot fix the type mismatch, and
- The analysis does not cover a significant portion of your code because calls via function pointers are not recognized.

Settings

☑ On

The verification must allow function pointer calls where the type of the function pointer does not match the type of the function. For instance, a function declared as `int f(int*)` can be called by a function pointer declared as `int (*fptr)(void*)`.

Only type mismatches between pointer types are allowed. Type mismatches between nonpointer types cause compilation errors. For instance, a function declared as `int f(int)` cannot be called by a function pointer declared as `int (*fptr)(double)`.

☐ Off (default)

The verification must require that the argument and return types of a function pointer and the function it calls are identical.

Type mismatches are detected with the check `Correctness condition`.

Tips

- With sources that use function pointers extensively, enabling this option can cause loss in performance. This loss occurs because the verification has to consider more execution paths.
- Using this option can increase the number of orange checks. Some of these orange checks can reveal a real issue with the code.

  Consider these examples where a type mismatch occurs between the function pointer type and the function that it points to:

  - In this example, the function pointer `obj_fptr` has an argument that is a pointer to a three-element array. However, it points to a function whose corresponding argument is a pointer to a four-element array. In the body of `foo`, four array elements are read and incremented. The fourth element does not exist and the `++` operation reads a meaningless value.

    ```
    typedef int array_three_elements[3];
    typedef void (*fptr)(array_three_elements*);

    typedef int array_four_elements[4];
    void foo(array_four_elements*);

    void main() {
     array_three_elements arr[3] = {0,0,0};
     array_three_elements *ptr;
     fptr obj_fptr;

     ptr = &arr;
     obj_fptr = &foo;

     //Call via function pointer
     obj_fptr(&ptr);
    }

    void foo(array_four_elements* x) {
        int i = 0;
        int *current_pos;

        for(i = 0; i< 4; i++) {
          current_pos = (*x) + i;
          (*current_pos)++;
        }
    }
    ```

    Without this option, an orange `Correctness condition` check appears on the call `obj_fptr(&ptr)` and the function `foo` is not verified. If you use this option, the body of `foo` contains several orange checks. Review the checks carefully and make sure that the type mismatch does not cause issues.

  - In this example, the function pointer has an argument that is a pointer to a structure with three `float` members. However, the corresponding function argument is a pointer to an unrelated structure with one array member. In the function body, the `strlen` function is used assuming the array member. Instead the `strlen` call reads the `float` members and can read meaningless values, for instance, values stored in the structure padding.

```
#include <string.h>
struct point {
  float x;
  float y;
  float z;
};
struct message {
  char msg[10] ;
};
void foo(struct message*);

void main() {
  struct point pt = {3.14, 2048.0, -1.0} ;
  void (*obj_fptr)(struct point *) ;

  obj_fptr = &foo;

  //Call via function pointer
  obj_fptr(&pt);
}

void foo(struct message* x) {
  int y = strlen(x->msg) ;
}
```

Without this option, an orange `Correctness condition` check appears on the call `obj_fptr(&pt)` and the function `foo` is not verified. If you use this option, the function contains an orange check on the `strlen` call. Review the check carefully and make sure that the type mismatch does not cause issues.

Dependency

This option is available only if you set `Source code language (-lang)` to `C`.

Command-Line Information

Parameter: `-permissive-function-pointer`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-lang c -permissive-function-pointer`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-lang c -permissive-function-pointer`

See Also

`Correctness condition`

Topics
“Specify Polyspace Analysis Options”

Overflow mode for signed integer (`-signed-integer-overflows`)

Specify whether result of overflow is wrapped around or truncated

Description

This option affects a Code Prover analysis only.

Specify whether Polyspace flags signed integer overflows and whether the analysis wraps the result of an overflow or restricts it to its extremum value.

Set Option

User interface (desktop products only): In the **Configuration** pane, the option is on the **Check Behavior** node under **Code Prover Verification**.

Command line: Use the option `-signed-integer-overflows`. See “Command-Line Information” (Polyspace Code Prover).

Why Use This Option

Use this option to specify whether to check for signed integer overflows and to specify the assumptions the analysis makes following an overflow.

Settings

Default: `forbid`

`forbid`

Polyspace flags signed integer overflows. If the **Overflow** check on an operation is:

- Red, Polyspace does not analyze the remaining code in the current scope.
- Orange, Polyspace analyzes the remaining code in the current scope. Polyspace considers that:
  - After a positive **Overflow**, the result of the operation has an upper bound. This upper bound is the maximum value allowed by the type of the result.
  - After a negative **Overflow**, the result of the operation has a lower bound. This lower bound is the minimum value allowed by the type of the result.

This behavior conforms to the ANSI C (ISO C++) standard.

In the following code, `j` has values in the range `[1..`2^{31}`-1]` before the orange overflow. Polyspace considers that `j` has even values in the range `[2 .. 2147483646]` after the overflow. Polyspace does not analyze the `printf()` statement after the red overflow.

```
#include<stdio.h>

int getVal();

void func1()
{
    int i = 1;
    i = i << 30;
    // Result of * operation overflows
    i = i * 2;
    // Remaing code in current scope not analyzed
    printf("%d", i);
}
void func2()
{

    int j = getVal();
    if (j > 0) {
        // Range of j: [1..2^31-1]
        // Result of * operation may overflow
        j = j * 2;
        // Range of j: even values in [2 .. 2147483646]
        printf("%d", j);
    }
}
```

`allow`

Polyspace does not flag signed integer overflows. If an operation results in an overflow, Polyspace analyzes the remaining code but wraps the result of the overflow.

In this code, the analysis does not flag any overflow in the code. However, the range of `j` wraps around to even values in the range `[-2^31..2] or [2..2^31-2]` and the value of `i` wraps around to -2^{31}.

```
#include<stdio.h>

int getVal();

void func1()
{
    int i = 1;
    i = i << 30;
    // i = 2^30
    i = i * 2;
    // i = -2^31
    printf("%d", i);
}
void func2()
{

    int j = getVal();
    if (j > 0) {
        // Range of j: [1..2^31-1]
        j = j * 2;
        // Range of j: even values in [-2^31..2] or [2..2^31-2]
        printf("%d", j);
    }
}
```

`warn-with-wrap-around`

Polyspace flags signed integer overflows. If an operation results in an overflow, Polyspace analyzes the remaining code but wraps the result of the overflow.

In the following code, `j` has values in the range `[1..2^31-1]` before the orange overflow. Polyspace considers that `j` has even values in the range `[-2^31..2] or [2..2^31-2]` after the overflow.

Similarly, `i` has value 2^{30} before the red overflow and value -2^{31} after it .

```
#include<stdio.h>

int getVal();

void func1()
{
    int i = 1;
    i = i << 30;
    // i = 2^30
    // Result of * operation overflows
    i = i * 2;
    // i = -2^31
    printf("%d", i);
}
void func2()
{

    int j = getVal();
    if (j > 0) {
        // Range of j: [1..2^31-1]
        // Result of * operation may overflow
        j = j * 2;
        // Range of j: even values in [-2^31..2] or [2..2^31-2]
        printf("%d", j);
    }
}
```

Tips

- To check for overflows on conversions from unsigned to signed integers of the same size, set **Overflow mode for unsigned integer** to `forbid` or `warn-with-wrap-around`. If you allow unsigned integer overflows, Polyspace does not flag overflows on conversions and wraps the result of an overflow, even if you check for signed integer overflows.
- In Polyspace Code Prover, overflowing signed constants are wrapped around. This behavior cannot be changed by using the options. If you want to detect overflows with signed constants, use the Polyspace Bug Finder checker `Integer constant overflow`.

Command-Line Information

Parameter: `-signed-integer-overflows`
Value: `forbid` | `allow` | `warn-with-wrap-around`
Default: `forbid`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-signed-integer-overflows allow`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-signed-integer-overflows allow`

See Also

`Overflow` | `Overflow mode for unsigned integer (-unsigned-integer-overflows)`

Topics
“Specify Polyspace Analysis Options”

Introduced in R2018b

Overflow mode for unsigned integer (-unsigned-integer-overflows)

Specify whether result of overflow is wrapped around or truncated

Description

This option affects a Code Prover analysis only.

Specify whether Polyspace flags unsigned integer overflows and whether the analysis wraps the result of an overflow or restricts it to its extremum value.

Set Option

User interface (desktop products only): In the **Configuration** pane, the option is on the **Check Behavior** node under **Code Prover Verification**.

Command line: Use the option `-unsigned-integer-overflows`. See “Command-Line Information” (Polyspace Code Prover).

Why Use This Option

Use this option to specify whether to check for unsigned integer overflows and to specify the assumptions the analysis makes following an overflow.

Settings

Default: `allow`

`forbid`

Polyspace flags unsigned integer overflows. If the **Overflow** check on an operation is:

- Red, Polyspace does not analyze the remaining code in the current scope.
- Orange, Polyspace analyzes the remaining code in the current scope. Polyspace considers that:
  - After a positive **Overflow**, the result of the operation has an upper bound. This upper bound is the maximum value allowed by the type of the result.
  - After a negative **Overflow**, the result of the operation has a lower bound. This lower bound is the minimum value allowed by the type of the result.

In the following code, `j` has values in the range `[1..`2^{32}`-1]` before the orange overflow. Polyspace considers that `j` has even values in the range `[2 .. 4294967294]` after the overflow. Polyspace does not analyze the `printf()` statement after the red overflow.

```
#include<stdio.h>

unsigned int getVal();

void func1()
{
    unsigned int i = 1;
    i = i << 31;
    // Result of * operation overflows
    i = i * 2;
    // Remaing code in current scope not analyzed
    printf("%u", i);
}
void func2()
{

    unsigned int j = getVal();
    if (j > 0) {
        // Range of j: [1..2^32-1]
        // Result of * operation may overflow
        j = j * 2;
        // Range of j: even values in [2 .. 4294967294]
        printf("%u", j);
    }
}
```

`allow`

Polyspace does not flag unsigned integer overflows. If an operation results in an overflow, Polyspace analyzes the remaining code but wraps the result of the overflow. For instance, `MAX_INT + 1` wraps to `MIN_INT`. This behavior conforms to the ANSI C (ISO C++) standard.

In this code, the analysis does not flag any overflow in the code. However, the range of `j` wraps around to even values in the range `[0..2^32-2]]` and the value of `i` wraps around to `0`.

```
#include<stdio.h>

unsigned int getVal();

void func1()
{
    unsigned int i = 1;
    i = i << 31;
    // i = 2^31
    i = i * 2;
    // i = 0
    printf("%u", i);
}
void func2()
{

    unsigned int j = getVal();
    if (j > 0) {
        // Range of j: [1..2^32-1]
        j = j * 2;
        // Range of j: even values in [0 .. 4294967294]
        printf("%u", j);
    }
}
```

`warn-with-wrap-around`

Polyspace flags unsigned integer overflows. If an operation results in an overflow, Polyspace analyzes the remaining code but wraps the result of the overflow. For instance, `MAX_INT + 1` wraps to `MIN_INT`.

In the following code, `j` has values in the range $[1..2^{32}-1]$ before the orange overflow. Polyspace considers that `j` has even values in the range `[0 .. 4294967294]` after the overflow.

Similarly, `i` has value 2^{31} before the red overflow and value `0` after it.

```
#include<stdio.h>

unsigned int getVal();

void func1()
{
    unsigned int i = 1;
    i = i << 31;
    // i = 2^31
    i = i * 2;
    // i = 0
    printf("%u", i);
}
void func2()
{

    unsigned int j = getVal();
    if (j > 0) {
        // Range of j: [1..2^32-1]
        j = j * 2;
        // Range of j: even values in [0 .. 4294967294]
        printf("%u", j);
    }
}
```

Tips

- To check for overflows on conversions from unsigned to signed integers of the same size, set **Overflow mode for unsigned integer** to `forbid` or `warn-with-wrap-around`. If you allow unsigned integer overflows, Polyspace does not flag overflows on conversions and wraps the result of an overflow, even if you check for signed integer overflows.
- In Polyspace Code Prover, overflowing unsigned constants are wrapped around. This behavior cannot be changed by using the options. If you want to detect overflows with unsigned constants, use the Polyspace Bug Finder checker `Unsigned integer constant overflow`.

Command-Line Information

Parameter: `-unsigned-integer-overflows`
Value: `forbid` | `allow` | `warn-with-wrap-around`
Default: `allow`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-unsigned-integer-overflows allow`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-unsigned-integer-overflows allow`

See Also

`Overflow` | `Overflow mode for signed integer (-signed-integer-overflows)`

Topics
“Specify Polyspace Analysis Options”

Introduced in R2018b

Allow incomplete or partial allocation of structures (-size-in-bytes)

Allow a pointer with insufficient memory buffer to point to a structure

Description

This option affects a Code Prover analysis only.

Specify that the verification must allow dereferencing a pointer that points to a structure but has a sufficient buffer for only some of the structure's fields.

This type of pointer results when a pointer to a smaller structure is cast to a pointer to a larger structure. The pointer resulting from the cast has sufficient buffer for only some fields of the larger structure.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-size-in-bytes`. See "Command-Line Information" on page 1-272.

Why Use This Option

Use this option to relax the check for illegally dereferenced pointers. You can point to a structure even when the buffer allowed for the pointer is not sufficient for all the structure fields.

Settings

☑ On

When a pointer with insufficient buffer is dereferenced,Polyspace does not produce an **Illegally dereferenced pointer** error, as long as the dereference occurs within allowed buffer.

For instance, in the following code, the pointer `p` has sufficient buffer for the first two fields of the structure `BIG`. Therefore, with the option on, Polyspace considers that the first two dereferences are valid. The third dereference takes `p` outside its allowed buffer. Therefore, Polyspace produces an **Illegally dereferenced pointer** error on the third dereference.

```
#include <stdlib.h>

typedef struct _little { int a; int b; } LITTLE;
typedef struct _big { int a; int b; int c; } BIG;

void main(void) {
   BIG *p = malloc(sizeof(LITTLE));

   if (p!= ((void *) 0) ) {
      p->a = 0 ;
      p->b = 0 ;
      p->c = 0 ;   // Red IDP check
```

```
    }
}
```

Off (default)

Polyspace does not allow dereferencing a pointer to a structure if the pointer does not have sufficient buffer for all fields of the structure. It produces an **Illegally dereferenced pointer** error the first time you dereference the pointer.

For instance, in the following code, even though the pointer `p` has sufficient buffer for the first two fields of the structure `BIG`, Polyspace considers that dereferencing `p` is invalid.

```
#include <stdlib.h>

typedef struct _little { int a; int b; } LITTLE;
typedef struct _big { int a; int b; int c; } BIG;

void main(void) {
   BIG *p = malloc(sizeof(LITTLE));

   if (p!= ((void *) 0) ) {
      p->a = 0 ;   // Red IDP check
      p->b = 0 ;
      p->c = 0 ;
   }
}
```

Tips

- If you do not turn on this option, you cannot point to the field of a partially allocated structure.

  For instance, in the preceding example, if you do not turn on the option and perform the assignment

  ```
  int *ptr = &(p->a);
  ```

  Polyspace considers that the assignment is invalid. If you dereference `ptr`, it produces an **Illegally dereferenced pointer** error.
- Using this option can slightly increase the number of orange checks.

Command-Line Information

Parameter: `-size-in-bytes`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-size-in-bytes`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-size-in-bytes`

See Also

`Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)` | `Illegally dereferenced pointer`

Topics
“Specify Polyspace Analysis Options”

Subnormal detection mode (-check-subnormal)

Detect operations that result in subnormal floating-point values

Description

This option affects a Code Prover analysis only.

Specify that the verification must check floating-point operations for subnormal results.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-check-subnormal`. See “Command-Line Information” on page 1-275.

Why Use This Option

Use this option to detect floating-point operations that result in subnormal values.

Subnormal numbers have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the significand. The presence of subnormal numbers indicates loss of significant digits. This loss can accumulate over subsequent operations and eventually result in unexpected values. Subnormal numbers can also slow down the execution on targets without hardware support.

Settings

Default: `allow`

`allow`

The verification does not check operations for subnormal results.

`forbid`

The verification checks for subnormal results.

The verification stops the execution path with the subnormal result and prevents subnormal values from propagating further. Therefore, in practice, you see only the first occurrence of the subnormal value.

`warn-all`

The verification checks for subnormal results and highlights all occurrences of subnormal values. Even if a subnormal result comes from previous subnormal values, the result is highlighted.

The verification continues even if the check is red.

`warn-first`

The verification checks for subnormal results but only highlights first occurrences of subnormal values. If a subnormal value propagates to further subnormal results, those subsequent results are not highlighted.

The verification continues even if the check is red.

For details of the result colors in each mode, see `Subnormal float`.

Tips

- If you want to see only those operations where a subnormal value originates from non-subnormal operands, use the `warn-first` mode.

  For instance, in the following code, `arg1` and `arg2` are unknown. The verification assumes that they can take all values allowed for the type `double`. This assumption can lead to subnormal results from certain operations. If you use the `warn-first` mode, the first operation causing the subnormal result is highlighted.

warn-all	**warn-first**

`void func (double arg1, double arg2)`
`{`
`  double difference1 = arg1 - arg2;`
`  double difference2 = arg1 - arg2;`
`  double val1 = difference1 * 2;`
`  double val2 = difference2 * 2;`
`}`
 	
In this example, all four operations can have subnormal results. The four checks for subnormal results are orange.
	`void func (double arg1, double arg2)`
	`{`
	`  double difference1 = arg1 - arg2;`
	`  double difference2 = arg1 - arg2;`
	`  double val1 = difference1 * 2;`
	`  double val2 = difference2 * 2;`
	`}`
	
	In this example, `difference1` and `difference2` can be subnormal if `arg1` and `arg2` are sufficiently close. The first two checks for subnormal results are orange. `val1` and `val2` cannot be subnormal unless `difference1` and `difference2` are subnormal. The last two checks for subnormal results are green.
	
	Through red/orange checks, you see only the first instance where a subnormal value appears. You do not see red/orange checks from those subnormal values propagating to subsequent operations.

- If you want to see where a subnormal value originates and do not want to see subnormal results arising from the same cause more than once, use the `forbid` mode.

  For instance, in the following code, `arg1` and `arg2` are unknown. The verification assumes that they can take all values allowed for the type `double`. This assumption can lead to subnormal results for `arg1-arg2`. If you use the `forbid` mode and perform the operation `arg1-arg2` twice in succession, only the first operation is highlighted. The second operation is not highlighted because the subnormal result for the second operation arises from the same cause as the first operation.

warn-all	forbid

`void func (double arg1, double arg2)`
`{`
`  double difference1 = arg1 - arg2;`
`  double difference2 = arg1 - arg2;`
`  double val1 = difference1 * 2;`
`  double val2 = difference2 * 2;`
`}`
 	
In this example, all four operations can have subnormal results. The four checks for subnormal results are orange.
	`void func (double arg1, double arg2)`
	`{`
	`  double difference1 = arg1 - arg2;`
	`  double difference2 = arg1 - arg2;`
	`  double val1 = difference1 * 2;`
	`  double val2 = difference2 * 2;`
	`}`
	
	In this example, `difference1` can be subnormal if `arg1` and `arg2` are sufficiently close. The first check for subnormal results is orange. Following this check, the verification excludes from consideration:
	• The close values of `arg1` and `arg2` that led to the subnormal value of `difference1`.
	In the subsequent operation `arg1 - arg2`, the check is green and `difference2` is not subnormal. The result of the check on `difference2 * 2` is green for the same reason.
	• The subnormal value of `difference1`.
	In the subsequent operation `difference1 * 2`, the check is green.

- You cannot run the Automatic Orange Tester if you check for subnormals in your verification.

Command-Line Information

Parameter: `-check-subnormal`
Value: `allow` | `warn-first` | `warn-all` | `forbid`
Default: `allow`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-check-subnormal forbid`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-check-subnormal forbid`

See Also

Polyspace Results
`Subnormal float`

Topics
“Specify Polyspace Analysis Options”

Introduced in R2016b

Detect uncalled functions (`-uncalled-function-checks`)

Detect functions that are not called directly or indirectly from `main` or another entry point function

Description

This option affects a Code Prover analysis only.

Detect functions that are not called directly or indirectly from `main` or another entry point function during run-time.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

Command line: Use the option `-uncalled-function-checks`. See “Command-Line Information” on page 1-276.

Why Use This Option

Typically, after verification, the **Dashboard** pane shows functions that are not called during verification. However, you do not see them in your analysis results or reports. You cannot comment on them or justify them.

If you want to see these uncalled functions in your analysis results and reports, use this option.

Settings

Default: `none`

`none`
: The verification does not generate checks for uncalled functions.

`never-called`
: The verification generates checks for functions that are defined but not called.

`called-from-unreachable`
: The verification generates checks for functions that are defined and called from an unreachable part of the code.

`all`
: The verification generates checks for functions that are:

- Defined but not called
- Defined and called from an unreachable part of the code.

Command-Line Information

Parameter: `-uncalled-function-checks`

Value: `none` | `never-called` | `called-from-unreachable` | `all`
Default: `none`
Example (Code Prover): `polyspace-code-prover -sources file_name -uncalled-function-checks all`
Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -uncalled-function-checks all`

See Also

`Function not called` | `Function not reachable`

Topics
“Specify Polyspace Analysis Options”

Sensitivity context (`-context-sensitivity`)

Store call context information to identify function call that caused errors

Description

This option affects a Code Prover analysis only.

Specify the functions for which the verification must store call context information. If the function is called multiple times, using this option helps you to distinguish between the different calls.

Set Option

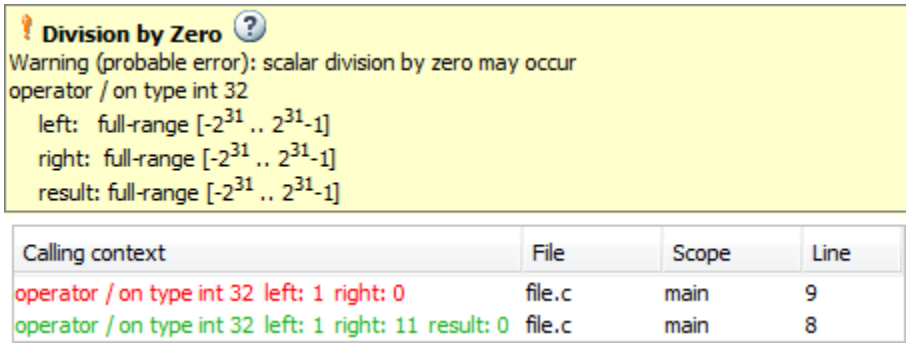
User interface (desktop products only): In your project configuration, the option is available on the **Precision** node.

Command line: Use the option `-context-sensitivity`. See “Command-Line Information” (Polyspace Code Prover).

Why Use This Option

Suppose a function is called twice in your code. The check color on each operation in the function body is a combined result of both calls. If you want to distinguish between the colors in the two calls, use this option.

For instance, if a function contains a red or orange check and a green check on the same operation for two different calls, the software combines the contexts and displays an orange check on the operation. If you use this option, the check turns dark orange and the result details show the color of the check for each call.



For a tutorial on using this option, see “Identify Function Call with Run-Time Error” (Polyspace Code Prover).

Settings

Default: none

none

The software does not store call context information for functions.

`auto`

The software stores call context information for checks in:

- Functions that form the leaves of the call tree. These functions are called by other functions, but do not call functions themselves.
- Small functions. The software uses an internal threshold to determine whether a function is small.

`custom`

The software stores call context information for functions that you specify. To enter the name of a function, click  .

Tips

- If you select this option, you do not see tooltips in the body of the functions that benefit from this option (and keep the call contexts separate).
- If you select this option, the analysis can show some code operations in grey (unreachable code) even when you can identify execution paths leading to the operations. In this case, the grey code indicates operations that might be unreachable only in a particular call context.

  For instance, suppose this function is called with the arguments -1 and 1 :

  ```
  int isPositive (int num) {
     if(num < 0)
        return 0;
     return 1;
  }
  ```

  If you use the option with this function as argument, there are two unreachable code checks:

  - The check on `if` is grey because when the function is called with argument -1, the `if` condition is always true.
  - The check on the code inside the `if` branch is grey because when the function is called with argument 1, the `if` condition is always false.

  Each unreachable code check indicates code that is unreachable only in a particular call context. You see the call context in the result details.

Command-Line Information

Parameter: `-context-sensitivity`
Value: *`function1`*`[,`*`function2`*`,...]`
Default: `none`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-context-sensitivity myFunc1,myFunc2`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-context-sensitivity myFunc1,myFunc2`

To allow the software to determine which functions receive call context storage, use the option `-context-sensitivity-auto`.

See Also

Topics
“Specify Polyspace Analysis Options”
“Identify Function Call with Run-Time Error” (Polyspace Code Prover)

Improve precision of interprocedural analysis (-path-sensitivity-delta)

Avoid certain verification approximations for code with fewer lines

Description

This option affects a Code Prover analysis only.

For smaller code, use this option to improve the precision of cross-functional analysis.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Precision** node.

Command line: Use the option `-path-sensitivity-delta`. See “Command-Line Information” on page 1-281.

Why Use This Option

Use this option to avoid certain software approximations on execution paths. Avoiding these approximations results in fewer orange checks but a much longer verification time.

For instance, for deep function call hierarchies or nested conditional statements, to complete verification in a reasonable amount of time, the software combines many execution paths and stores less information at each stage of verification. If you use this option, the software stores more information about the execution paths, resulting in a more precise verification.

Settings

Default: Off

Enter a positive integer to turn on this option.

Entering a higher value leads to a greater number of proven results, but also increases verification time exponentially. For instance, a value of 10 can result in very long verification times.

Tips

Use this option only when you have less than 1000 lines of code.

Command-Line Information

Parameter: `-path-sensitivity-delta`
Value: Positive integer
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-path-sensitivity-delta 1`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-path-sensitivity-delta 1`

See Also

Topics
“Specify Polyspace Analysis Options”
“Improve Verification Precision” (Polyspace Code Prover)

Precision level (-0)

Specify a precision level for the verification

Description

This option affects a Code Prover analysis only.

Specify the precision level that the verification must use.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Precision** node.

Command line: Use the option `-0#`, for instance, `-00` or `-01`. See “Command-Line Information” on page 1-284.

Why Use This Option

Higher precision leads to greater number of proven results but also requires more verification time. Each precision level corresponds to a different algorithm used for verification.

In most cases, you see the optimal balance between precision and verification time at level 2.

Settings

Default: 2

`0`

This option corresponds to a static interval verification.

`1`

This option corresponds to a more complex static interval verification.

`2`

This option corresponds to a complex polyhedron model of domain values with additional precision for interprocedural analysis depending on the option `Improve precision of interprocedural analysis (-path-sensitivity-delta)`.

`3`

This option is only suitable for code having less than 1000 lines. Using this option, the percentage of proven results can be very high.

Tips

- For best results in reasonable time, use the default level 2. If the verification takes a long time, reduce precision. However, the number of unproven checks can increase. Likewise, to reduce orange checks, you can improve your precision. But the verification can take significantly longer time.

- The precision levels 2 and below begin to take effect only from verification levels higher than `Software Safety Analysis level 0`. See also `Verification level (-to)`.

  For instance, to reduce analysis time, you might have reduced the verification level to `Software Safety Analysis level 0`. Do not try to reduce the precision level below 2 to lower the analysis time further.

  Note that algorithms used in precision level 3 can also apply to the verification level `Software Safety Analysis level 0`.

Command-Line Information

Parameter: `-O0` | `-O1` | `-O2` | `-O3`
Default: `-O2`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-O1`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-O1`

See Also

`Specific precision (-modules-precision)` | `Verification level (-to)`

Topics

“Specify Polyspace Analysis Options”
“Improve Verification Precision” (Polyspace Code Prover)

Specific precision (-modules-precision)

Specify source files you want to verify at higher precision than the remaining verification

Description

This option affects a Code Prover analysis only.

Specify source files that you want to verify at a precision level higher than that for the entire verification.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Precision** node. See “Dependency” on page 1-285 for other options you must also enable.

Command line: Use the option `-modules-precision`. See “Command-Line Information” on page 1-285.

Why Use This Option

If a specific file is verified imprecisely leading to many orange checks in the file and elsewhere, you can improve the precision for that file.

Note that increasing precision also increases verification time.

Settings

Default: All files are verified with the precision you specified using **Precision > Precision level**.

Click  to enter the name of a file without the extension `.c` and the corresponding precision level.

Dependency

This option is available only if you set `Source code language (-lang)` to `C` or `C-CPP`.

Command-Line Information

Parameter: `-modules-precision`
Value: *`file`*`:O0` | *`file`*`:O1` | *`file`*`:O2` | *`file`*`:O3`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-O1 -modules-precision My_File:02`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-O1 -modules-precision My_File:02`

See Also

`Precision level (-O)`

Topics

“Specify Polyspace Analysis Options”

“Improve Verification Precision” (Polyspace Code Prover)

Verification level (-to)

Specify number of times the verification process runs on your code

Description

This option affects a Code Prover analysis only.

Specify the number of times the Polyspace verification process runs on your source code. Each run can lead to greater number of proven results but also requires more verification time.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Precision** node.

Command line: Use the option `-to`. See “Command-Line Information” on page 1-289.

Why Use This Option

There are many reasons you might want to increase or decrease the verification level. For instance:

- Coding rules are checked early during the compilation phase, with some exception only. If you check for coding rules alone, you can lower the verification level. See “Check for Coding Standard Violations”.
- If you see many orange checks after verification, try increasing the verification level. However, increasing the verification level also increases verification time.

  In most cases, you see the optimal balance between precision and verification time at level 2.

Settings

Default: `Software Safety Analysis level 2`

`Source Compliance Checking`

Polyspace checks for compilation errors only. Most coding rule violations are also found in this phase.

`Software Safety Analysis level 0`

The verification process performs some simple analysis. The analysis is designed to reach completion despite complexities in the code.

If the verification gets stuck at a higher level, try running to this level and review the results.

`Software Safety Analysis level 1`

The verification process analyzes each function once with algorithms whose complexity depends on the precision level. See `Precision level (-O)`. The analysis starts from the top of the function call hierarchy (an actual or generated `main` function) and propagates to the leaves of the call hierarchy.

`Software Safety Analysis level 2`

The verification process analyzes each function twice. In the first pass, the analysis propagates from the top of the function call hierarchy to the leaves. In the second pass, the analysis

propagates from the leaves back to the top. Each pass uses information gathered from the previous pass.

Use this option for most accurate results in reasonable time.

`Software Safety Analysis level 3`

The verification process runs three times on each function: from the top of the function call hierarchy to the leaves, from the leaves to the top, and from the top to the leaves again. Each pass uses information gathered from the previous pass.

`Software Safety Analysis level 4`

The verification process runs four passes on each function: from the top of the function call hierarchy to the leaves twice. Each pass uses information gathered from the previous pass.

`other`

If you use this option, Polyspace verification will make 20 passes unless you stop it manually.

Tips

- Use a higher verification level for fewer orange checks.

  In some cases, if the verification can detect that results of maximum precision are available after an earlier level, the verification stops and does not proceed to the level that you specify.

  Difference between Level 0 and 1

  The following example illustrates the difference between `Software Safety Analysis level 0` and `Software Safety Analysis level 1`. In level 1, Code Prover can establish the success of the final assertion that involves a relation between two array elements even without knowing the actual elements of the array.

Software Safety Analysis Level 0	Software Safety Analysis Level 1

`extern int tab[];`
 	
`int main() {`
 	
`    int i = tab[3];`
`    int j = tab[1];`
 	
`    if (i > j) {`
`        int l = i-j;`
`        assert(l > 0);`
`     }`
`}`
	`extern int tab[];`
	
	`int main() {`
	
	`    int i = tab[3];`
	`    int j = tab[1];`
	
	`    if (i > j) {`
	`        int l = i-j;`
	`        assert(l > 0);`
	`     }`
	`}`

  In the table, verification produces an orange `Division by Zero` check during level 0 verification. The check turns green during level 1. The verification acquires more precise knowledge of `x` in the higher level.

  If a higher verification level fails because the verification runs out of memory, but results are available at a lower level, Polyspace displays the results from the lower level.

- For best results, use the option `Software Safety Analysis level 2`. If the verification takes too long, use a lower **Verification level**. Fix red errors and gray code before rerunning the verification with higher verification levels.

- Use the option `Other` sparingly since it can increase verification time by an unreasonable amount. Using `Software Safety Analysis level 2` provides optimal verification of your code in most cases.
- If the **Verification Level** is set to `Source Compliance Checking`, do not run verification on a remote server. The source compliance checking, or compilation, phase takes place on your local computer anyway. Therefore, if you are running verification only to the end of compilation, run verification on your local computer.
- If you want to see global variable sharing and usage only use to run a less extensive analysis.

Command-Line Information

Parameter: `-to`
Value: `compile` | `pass0` | `pass1` | `pass2` | `pass3` | `pass4` | `other`
Default: `pass2`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-to pass2`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-to pass2`

You can also use these additional values not available in the user interface:

- C projects: `c-to-il` (C to intermediate language conversion phase)
- C++ projects: `cpp-to-il` (C++ to intermediate language conversion phase), `cpp-normalize` (C++ normalization phase), `cpp-link` (C++ link phase)

Use these values only if you have specific reasons to do so. For instance, to generate a blank constraints (DRS) template for C++ projects, you have to run an analysis up to the `cpp-normalize` phase.

See Also

`Precision level (-O)`

Topics

“Specify Polyspace Analysis Options”
“Improve Verification Precision” (Polyspace Code Prover)

Verification time limit (`-timeout`)

Specify a time limit on your verification

Description

This option affects a Code Prover analysis only.

Specify a time limit for the verification in hours. If the verification does not complete within that limit, it stops.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Precision** node.

Command line: Use the option `-timeout`. See “Command-Line Information” on page 1-290.

Why Use This Option

Use this option to impose a time limit on the verification.

By default, if an internal step in the verification lasts for more than 24 hours, the verification stops. You can use this option to reduce the time limit even further. Note that you can have verification results despite the verification timing out. For instance, if a step in Software Safety Analysis level 1 times out, you still get the results from level 0. See `Verification level (-to)`.

The option is useful only in very specific cases. Suppose your code has certain constructs that might slow down the verification. To check this, you can impose a time limit on the verification so that the verification stops if it takes too long.

Typically, Technical Support asks you to use this option as needed.

Settings

Enter the time in hours. For fractions of an hour, specify decimal form.

Command-Line Information

Parameter: `-timeout`
Value: *`time`*
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-timeout 5.75`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-timeout 5.75`

See Also

Topics

“Specify Polyspace Analysis Options”
“Improve Verification Precision” (Polyspace Code Prover)

Inline (-inline)

Specify functions that must be cloned internally for each function call

Description

This option affects a Code Prover analysis only.

Specify the functions that the verification must clone internally for every function call.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Scaling** node.

Command line: Use the option `-inline`. See “Command-Line Information” on page 1-292.

Why Use This Option

Use this option sparingly. Sometimes, using the option helps to work around scaling issues during verification. If your verification takes too long, Technical Support can ask you to use this option for certain functions.

Do not use this option to understand results. For instance, suppose a function is called twice in your code. The check color on each operation in the function body is a combined result of both calls. If you want to distinguish between the colors in the two calls, use the option `Sensitivity context (-context-sensitivity)`.

Settings

No Default

Enter function names or choose from a list.

- Click  to add a field and enter the function name.
- Click  to list functions in your code. Choose functions from the list.

The verification internally clones the function for each call. For instance, if you specify the function `func` for inlining and `func` is called twice, the software creates two copies of `func` for verification. The copies are named using the convention `func_pst_inlined_`*`ver`* where *`ver`* is the version number. You see both copies on the **Call Hierarchy** pane.

However, for each run-time check in the function body, you see only one color in your verification results. The semantics of the check color is different from the normal specification.

Red checks:

- Normally, if a function is called twice and an operation causes a definite error only in one of the calls, the check color is orange.
- If you use this option, the color changes to dark orange (shown with an orange exclamation mark in the results list).

Gray checks:

- Normally, if a function is called twice and an `if` statement branch is unreachable in only one of the calls, the branch is shown as reachable.
- If you use this option, the worst color is shown for the check. Therefore, the `if` branch appears gray.

Do not use this option to understand results. Use this option only if a certain function causes scaling issues.

Tips

- Use this option to identify the cause of a **Non-terminating call** error.
  - **Situation:** Sometimes, a red **Non-terminating call** check can appear on a function call though a red check does not appear in the function body. The function body represents all calls to the function. Therefore, if some calls to a function do not cause an error, an orange check appears in the function body.
  - **Action:** If you use this option, for every function call, there is a corresponding function body. Therefore, you can trace a red check on a function call to a red check in the function body.
- Using this option can sometimes duplicate a lot of code and lead to scaling problems. Therefore choose functions to inline carefully.
- Choose functions to inline based on hints provided by the alias verification.
- Do not use this option for entry point functions, including `main`.
- Using this option can increase the number of gray **Unreachable code** checks.

  For example, in the following code, if you enter `max` for **Inline**, you obtain two **Unreachable code** checks, one for each call to `max`.

```
int max(int a, int b) {
  return a > b ? a : b;
}

void main() {
  int i=3, j=1, k;
  k=max(i,j);
  i=0;
  k=max(i,j);
}
```

- If you use the keyword `inline` before a function definition, place the definition in a header file and call the function from multiple source files, you have the same result as using the option **Inline**.
- For C++ code, this option applies to all overloaded methods of a class.

Command-Line Information

Parameter: `-inline`
Value: *`function1`*`[,`*`function2`*`[,...]]`
No Default
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-inline func1,func2`

Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-inline func1,func2`

See Also

Topics
“Specify Polyspace Analysis Options”

Depth of verification inside structures (-k-limiting)

Limit the depth of analysis for nested structures

Description

This option affects a Code Prover analysis only.

Specify a limit to the depth of analysis for nested structures.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Scaling** node.

Command line: Use the option `-k-limiting`. See “Command-Line Information” on page 1-294.

Why Use This Option

Use this option if the analysis is slow because your code has a structure that is many levels deep.

Typically, you see a warning message when a structure with a deep hierarchy is slowing down the verification.

Settings

Default: Full depth of nested structures is analyzed.

Enter a number to specify the depth of analysis for nested structures. For instance, if you specify 0, the analysis does not verify a structure inside a structure.

If you specify a number less than 2, the verification could be less precise.

Command-Line Information

Parameter: `-k-limiting`
Value: *`positive integer`*
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-k-limiting 3`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-k-limiting 3`

See Also

Topics

“Specify Polyspace Analysis Options”

Generate report

Specify whether to generate a report after the analysis

Description

Specify whether to generate a report along with analysis results.

Depending on the format you specify, you can view this report using an external software. For example, if you specify the format `PDF`, you can view the report in a `pdf` reader.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Reporting** node.

Command line: See “Command-Line Information” on page 1-296.

Why Use This Option

You can generate a report from your analysis results for archiving purposes. You can provide this report to your management or clients as proof of code quality.

Using other analysis options, you can tailor the report content and format for your specific needs. See `Bug Finder and Code Prover report (-report-template)` and `Output format (-report-output-format)`.

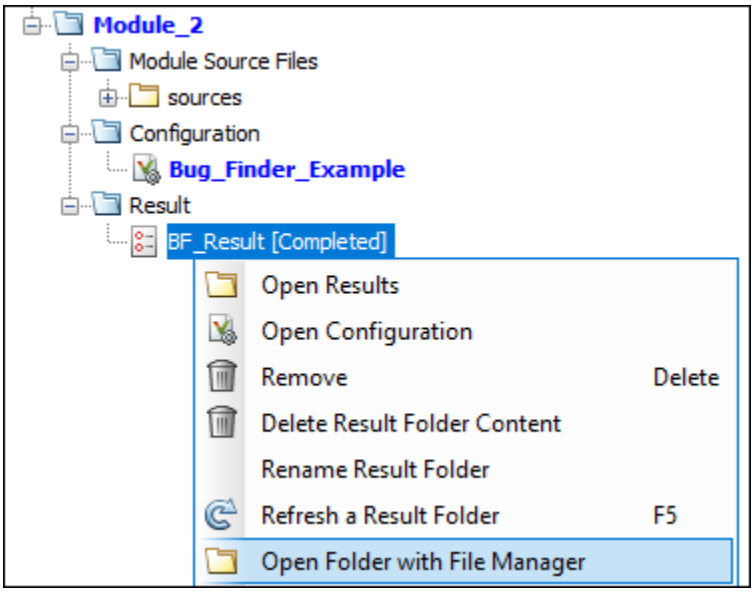
Settings

☑ On

Polyspace generates an analysis report using the template and format you specify.

The report is stored in the `Polyspace-Doc` subfolder of your results folder.

In Polyspace desktop products, to open your results folder from the user interface, on the **Project Browser** pane, right-click the results node and select **Open Folder with File Manager**.



To change the results folder location, see “Project and Results Folder Contents”.

On the command-line, the results folder is the argument of the option `-results-dir`.

Off (default)

Polyspace does not generate an analysis report. You can still view your results in the Polyspace interface.

Tips

This option allows you to specify report generation before starting an analysis.

To generate a report *after* an analysis is complete, in the user interface of the Polyspace desktop products, select **Reporting** > **Run Report**. Alternatively, at the command line, use the `polyspace-report-generator` command.

After analysis, you can also export the result as a text file for further customization. Use the option `-generate-results-list-file` with the `polyspace-report-generator` command.

Command-Line Information

There is no command-line option to solely turn on the report generator. However, using the options `-report-template` for template and `-report-output-format` for output format automatically turns on the report generator.

See Also

`Bug Finder and Code Prover report (-report-template)` | `Output format (-report-output-format)` | `polyspace-report-generator`

Topics

“Specify Polyspace Analysis Options”
“Generate Reports”

Bug Finder and Code Prover report (-report-template)

Specify template for generating analysis report

Description

Specify template for generating analysis report.

`.rpt` files for the report templates are available in *`polyspaceroot`*`\toolbox\polyspace\psrptgen\templates\`. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Reporting** node. You have separate options for Bug Finder and Code Prover analysis. See “Dependencies” on page 1-302 for other options you must also enable.

Command line: Use the option `-report-template`. See “Command-Line Information” on page 1-303.

Why Use This Option

Depending on the template that you use, the report contains information about certain types of results from the **Results List** pane. The template also determines what information is presented in the report and how the information is organized. See the template descriptions below.

Settings – Bug Finder

Default: `BugFinderSummary`

`BugFinder`

The report lists:

- **Polyspace Bug Finder Summary**: Number of results in the project. The results are summarized by file. The files that are partially analyzed because of compilation errors are listed in a separate table.
- **Code Metrics**: Summary of the various code complexity metrics. For more information, see “Code Metrics”.
- **Coding Rules**: Coding rule violations in the source code. For each rule violation, the report lists the:
  - Rule number and description.
  - Function containing the rule violation.
  - Review information, such as **Severity**, **Status** and comments.
- **Defects**: Defects found in the source code. For each defect, the report lists the:

- Function containing the defect.
- Defect information on the **Result Details** pane.
- Review information, such as **Severity**, **Status** and comments.

- **Configuration Settings**: List of analysis options that Polyspace uses for analysis. If you configured your project for multitasking, this section also lists the **Concurrency Modeling Summary**. If your project has source files with compilation errors, these files are also listed.

  If you check for coding rules, an additional **Coding Rules Configuration** section states the rules along with the information whether they were enabled or disabled.

`BugFinderSummary`

The report lists:

- **Polyspace Bug Finder Summary**: Number of results in the project. The results are summarized by file. The files that are partially analyzed because of compilation errors are listed in a separate table.
- **Code Metrics**: Summary of the various code complexity metrics. For more information, see “Code Metrics”.
- **Coding Rules Summary**: Coding rules along with number of violations.
- **Defect Summary**: Defects that Polyspace Bug Finder looks for. For each defect, the report lists the:

  - Defect group.
  - Defect name.
  - Number of instances of the defect found in the source code.

- **Configuration Settings**: List of analysis options that Polyspace uses for analysis. If you configured your project for multitasking, this section also lists the **Concurrency Modeling Summary**. For more information, see “Analysis Options”. If your project has source files with compilation errors, these files are also listed.

  If you check for coding rules, an additional **Coding Rules Configuration** section states the rules along with the information whether they were enabled or disabled.

`CodeMetrics`

The report lists the following:

- **Code Metrics Summary**: Various quantities related to the source code. For more information, see “Code Metrics”.
- **Code Metrics Details**: Various quantities related to the source code with the information broken down by file and function.
- **Configuration Settings**: List of analysis options that Polyspace uses for analysis. If you configured your project for multitasking, this section also lists the **Concurrency Modeling Summary**. If your project has source files with compilation errors, these files are also listed.

  If you check for coding rules, an additional **Coding Rules Configuration** section states the rules along with the information whether they were enabled or disabled.

`CodingStandards`

The report contains separate chapters for each coding standard enabled in the analysis (for instance, MISRA C: 2012, CERT C, custom rules, and so on). Each chapter contains the following information:

- **Summary - Violations by File**: Graph showing each file with number of rule violations.
- **Summary - Violations by Rule**: Graph showing each rule with number of violations. If a rule is not enabled or not violated, it does not appear in the graph.
- **Summary for all Files**: Table showing each file with number of rule violations.
- **Summary for Enabled Guidelines** or **Summary for Enabled Rules**: Table showing each guideline or rule with number of violations.
- **Violations**: Tables listing each rule violation, along with information such as ID, function name, severity, status, and so on. One table is created per file.

An appendix lists the options used in the Polyspace analysis.

`SecurityCWE`

The report contains the same information as the `BugFinder` report. However, in the **Defects** chapter, an additional column lists the CWE rules mapped to each defect. The **Configuration Settings** appendix also includes a **Security Standard to Polyspace Result Map**.

`Metrics`

Only available for results downloaded from the Polyspace Metrics interface.

The report lists information useful to quality engineers and available on the Polyspace Metrics interface, including:

- Information about whether the project satisfies quality objectives
- Time taken in each phase of analysis
- Metrics about the whole project. For each metric, the report lists the quality threshold and whether the metric satisfies this threshold.
- Coding rule violations in the project. For each rule, the report lists the number of violations justified and whether the justifications satisfy quality objectives.
- Definite as well as possible run-time errors in the project. For each type of run-time error, the report lists the number of errors justified and whether the justifications satisfy quality objectives.

The appendices contain further details of Polyspace configuration settings, code metrics, coding rule violations, and run-time errors.

Settings – Code Prover

Default: `Developer`

`CodeMetrics`

The report contains a summary of code metrics, followed by the complete metrics for an application.

`CodingStandards`

The report contains separate chapters for each coding standard enabled in the analysis (for instance, MISRA C: 2012, custom rules, and so on). Each chapter contains the following information:

- **Summary - Violations by File**: Graph showing each file with number of rule violations.
- **Summary - Violations by Rule**: Graph showing each rule with number of violations. If a rule is not enabled or not violated, it does not appear in the graph.
- **Summary for all Files**: Table showing each file with number of rule violations.
- **Summary for Enabled Guidelines** or **Summary for Enabled Rules**: Table showing each guideline or rule with number of violations.
- **Violations**: Tables listing each rule violation, along with information such as ID, function name, severity, status, and so on. One table is created per file.

An appendix lists the options used in the Polyspace analysis.

`Developer`

The report lists information useful to developers, including:

- Summary of results
- Coding rule violations
- List of proven run-time errors or red checks
- List of unproven run-time errors or orange checks
- List of unreachable procedures or gray checks
- Global variable usage in code. See “Global Variables” (Polyspace Code Prover).

The report also contains the Polyspace configuration settings and modifiable assumptions used in the analysis. If your project has source files with compilation errors, these files are also listed.

`DeveloperReview`

The report lists the same information as the `Developer` report. However, the reviewed results are sorted by severity and status, and unreviewed results are sorted by file location.

`Developer_withGreenChecks`

The report lists the same information as the `Developer` report. In addition, the report lists code proven to be error-free or green checks.

`Quality`

The report lists information useful to quality engineers, including:

- Summary of results
- Statistics about the code
- Graphs showing distributions of checks per file

The report also contains the Polyspace configuration settings and modifiable assumptions used in the analysis. If your project has source files with compilation errors, these files are also listed.

`VariableAccess`

The report displays the global variable access in your source code. The report first displays the number of global variables of each type. For information on the types, see “Global Variables” (Polyspace Code Prover). For each global variable, the report displays the following information:

- Variable name.

  The entry for each variable is denoted by |.
- Type of the variable.
- Number of read and write operations on the variable.
- Details of read and write operations. For each read or write operation, the table displays the following information:
  - File and function containing the operation in the form *`file_name.function_name`*.

    The entry for each read or write operation is denoted by `||`. Write operations are denoted by < and read operations by >.
  - Line and column number of the operation.

This report captures the information available on the **Variable Access** pane in the Polyspace user interface.

`CallHierarchy`

The report displays the call hierarchy in your source code. For each function call in your source code, the report displays the following information:

- Level of call hierarchy, where the function is called.

  Each level is denoted by |. If a function call appears in the table as `|||->` *`file_name.function_name`*, the function call occurs at the third level of the hierarchy. Beginning from `main` or an entry point, there are three function calls leading to the current call.
- File containing the function call.

  In addition, the line and column is also displayed.
- File containing the function definition.

  In addition, the line and column where the function definition begins is also displayed.

In addition, the report also displays uncalled functions.

This report captures the information available on the **Call Hierarchy** pane in the Polyspace user interface.

`SoftwareQualityObjectives`

The report lists information useful to quality engineers and available on the Polyspace Metrics interface, including:

- Information about whether the project satisfies quality objectives
- Time taken in each phase of verification
- Metrics about the whole project. For each metric, the report lists the quality threshold and whether the metric satisfies this threshold.
- Coding rule violations in the project. For each rule, the report lists the number of violations justified and whether the justifications satisfy quality objectives.
- Definite as well as possible run-time errors in the project. For each type of run-time error, the report lists the number of errors justified and whether the justifications satisfy quality objectives.

The appendices contain further details of Polyspace configuration settings, code metrics, coding rule violations, and run-time errors.

This template is available only if you generate a report from results uploaded to the Polyspace Access web interface or from results uploaded to the Polyspace Metrics web interface (and then downloaded to the Polyspace user interface) . In each case, you have to set the objectives explicitly in the web interface and then generate the reports.

`SoftwareQualityObjectives_Summary`

The report contains the same information as the `SoftwareQualityObjectives` report. However, it does not have the supporting appendices with details of code metrics, coding rule violations and run-time errors.

This template is available only if you generate a report from results uploaded to the Polyspace Access web interface or from results uploaded to the Polyspace Metrics web interface (and then downloaded to the Polyspace user interface). In each case, you have to set a quality objective level explicitly in the web interface and then generate the reports.

Dependencies

In the user interface of the Polyspace desktop products, this option is enabled only if you select the `Generate report` option.

Tips

- This option allows you to specify report generation before starting an analysis.

  To generate a report *after* an analysis is complete, in the user interface of the Polyspace desktop products, select **Reporting** > **Run Report**. Alternatively, at the command line, use the `polyspace-report-generator` command.

  After analysis, you can also export the result as a text file for further customization. Use the option `-generate-results-list-file` with the `polyspace-report-generator` command.
- In Bug Finder, the report does not contain the line or column number for a result. Use the report for archiving, gathering statistics and checking whether results have been reviewed and addressed (for certification purposes or otherwise). To review a result in your source code, use the Polyspace user interface or your IDE if you are using a Polyspace plugin.
- If you use the `SoftwareQualityObjectives_Summary` and `SoftwareQualityObjectives` templates to generate reports, the pass/fail status depends on whether you set the quality objectives level in Polyspace Metrics or Polyspace Access:
  - In Polyspace Access, the pass/fail status is determined based on all results. For instance, if you use the level SQO-4 which sets a threshold of 60% on orange overflow checks, your project has a **FAIL** status if the percentage of green and justified orange overflow checks is less than 60% of *all green and orange overflow checks*.
  - In Polyspace Metrics, the pass/fail status is determined based on a file-by-file basis. The overall status is **FAIL** if one of the files have a **FAIL** status. For instance, if you use the level SQO-4 which sets a threshold of 60% on orange overflow checks, your project has a **FAIL** status if the percentage of green and justified orange overflow checks *in any file* is less than 60% of green and orange overflow checks in that file.
- The first chapter of the reports contain a summary of the relevant results. You can enter a Pass/Fail status in that chapter for your project based on the summary. If you use the template

SoftwareQualityObjectives or SoftwareQualityObjectives_Summary, the status is automatically assigned based on your objectives and the verification results. For more information on enforcing objectives using Polyspace Metrics, see “Compare Metrics Against Software Quality Objectives”.

Command-Line Information

Parameter: -report-template
Value: Full path to *template*.rpt
Example (Bug Finder): polyspace-bug-finder -sources *file_name* -report-template *polyspaceroot*\toolbox\polyspace\psrptgen\templates\bug_finder\BugFinder.rpt
Example (Code Prover): polyspace-code-prover -sources *file_name* -report-template *polyspaceroot*\toolbox\polyspace\psrptgen\templates\Developer.rpt
Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -report-template *polyspaceroot*\toolbox\polyspace\psrptgen\templates\bug_finder\BugFinder.rpt
Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -report-template *polyspaceroot*\toolbox\polyspace\psrptgen\templates\Developer.rpt

See Also

Generate report | Output format (-report-output-format) | polyspace-report-generator

Topics

“Specify Polyspace Analysis Options”
“Generate Reports”

Output format (`-report-output-format`)

Specify output format of generated report

Description

Specify output format of analysis report.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Reporting** node. See “Dependencies” on page 1-304 for other options you must also enable.

Command line: Use the option `-report-output-format`. See “Command-Line Information” on page 1-305.

Why Use This Option

Use this option to specify whether you want a report in PDF, HTML or another format.

Settings

Default: `Word`

`HTML`
: Generate report in `.html` format

`PDF`
: Generate report in `.pdf` format

`Word`
: Generate report in `.docx` format.

Tips

- This option allows you to specify report generation before starting an analysis.

  To generate a report *after* an analysis is complete, in the user interface of the Polyspace desktop products, select **Reporting** > **Run Report**. Alternatively, at the command line, use the `polyspace-report-generator` command.

  After analysis, you can also export the result as a text file for further customization. Use the option `-generate-results-list-file` with the `polyspace-report-generator` command.
- If the table of contents or graphics in a `.docx` report appear outdated, select the content of the report and refresh the document. Use keyboard shortcuts **Ctrl+A** to select the content and **F9** to refresh it.

Dependencies

In the user interface of the Polyspace desktop products, this option is enabled only if you select the `Generate report` option.

Command-Line Information

Parameter: -report-output-format
Value: html | pdf | word
Default: word
Example (Bug Finder): polyspace-bug-finder -sources *file_name* -report-output-format pdf
Example (Code Prover): polyspace-code-prover -sources *file_name* -report-output-format pdf
Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -report-output-format pdf
Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -report-output-format pdf

See Also

Bug Finder and Code Prover report (-report-template) | Generate report | polyspace-report-generator

Topics

“Specify Polyspace Analysis Options”
“Generate Reports”

Run Bug Finder or Code Prover analysis on a remote cluster (-batch)

Enable batch remote analysis

Description

This option applies to the Polyspace desktop products only. The option is used to send the analysis from a desktop to a server (where the analysis runs using the Polyspace server products).

Specify that the analysis must be offloaded to a remote server.

To offload a Polyspace analysis, you need:

- Polyspace Bug Finder Server™ and/or Polyspace Code Prover Server, and MATLAB Parallel Server™ on the server.
- Polyspace Bug Finder and/or Polyspace Code Prover on the desktop.

See “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

Set Option

User interface: In your project configuration, the option is on the **Run Settings** node. You have separate options for a Bug Finder and a Code Prover analysis.

Command line: Use the option `-batch`. See “Command-Line Information” on page 1-307.

Why Use This Option

Use this option if you want the analysis to run on a remote cluster instead of your local desktop.

For instance, you can run remote analysis when:

- You want to shut down your local machine but not interrupt the analysis.
- You want to free execution time on your local machine.
- You want to transfer the analysis to a more powerful computer.

Settings

☑ On

Run batch analysis on a remote computer. In this remote analysis mode, the analysis is queued on a cluster after the compilation phase. Therefore, on your local computer, after the analysis is queued:

- If you are running the analysis from the Polyspace user interface, you can close the user interface.
- If you are running the analysis from the command line, you can close the command-line window.

You can manage the queue from the Polyspace Job Monitor. To use the Polyspace Job Monitor:

- In the Polyspace user interface, select **Tools > Open Job Monitor**. See “Send Polyspace Analysis from Desktop to Remote Servers”.
- On the DOS or UNIX® command line, use the `polyspace-jobs-manager` command. For more information, see “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”.
- On the MATLAB command line, use the `polyspaceJobsManager` function.

After the analysis, you might have to manually download the results from the cluster.

☐ Off (default)

Do not run batch analysis on a remote computer.

Dependencies

- If you use a third-party scheduler instead of the MATLAB Job Scheduler, add the option `-no-credentials-check`. The credentials check performed in the product is only compatible with the MATLAB Job Scheduler. In the Polyspace user interface, add this option to the `Other` field.
- Do not run a Code Prover analysis on a remote cluster if you run up to the **Verification Level** of `Source Compliance Checking`. For both local and remote analysis, the source compliance checking or compilation phase takes place on your local computer. Therefore, if you are running only up to this phase, run on your local computer.

Command-Line Information

To run a remote analysis from the command line, use with the `-scheduler` option.
Parameter: `-batch`
Value: `-scheduler` *`host_name`* if you have not set the **Job scheduler host name** in the Polyspace user interface
Default: Off
Example (Bug Finder):
```
polyspace-bug-finder -batch -scheduler NodeHost
polyspace-bug-finder -batch -scheduler MJSName@NodeHost
```
Example (Code Prover):
```
polyspace-code-prover -batch -scheduler NodeHost
polyspace-code-prover -batch -scheduler MJSName@NodeHost
```

See Also

`-scheduler`

Topics

“Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”
“Specify Polyspace Analysis Options”
“Send Polyspace Analysis from Desktop to Remote Servers”
“Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”

Upload results to Polyspace Metrics (-add-to-results-repository)

Upload analysis results for viewing on Polyspace Metrics web dashboard

Description

This option applies to the Polyspace desktop products only.

Specify upload of analysis results to the Polyspace Metrics results repository, allowing Web-based reporting of results and code metrics.

Set Option

User interface: In your project configuration, the option is on the **Run Settings** node. You have separate options for a Bug Finder and a Code Prover analysis. See “Dependencies” on page 1-308 for other options that you must also enable.

Command line: Use the option `-add-to-results-repository`. See “Command-Line Information” on page 1-309.

Why Use This Option

Polyspace Metrics is a web dashboard that generates code quality metrics from your analysis results. Using this dashboard, you can:

- Provide your management a high-level overview of your code quality.
- Compare your code quality against predefined standards.
- Establish a process where you review in detail only those results that fail to meet standards.
- Track improvements or regression in code quality over time.

See “Generate Code Quality Metrics with Polyspace Metrics”.

Settings

☑ On

Analysis results are stored in the Polyspace Metrics results repository. This allows you to use a Web browser to view results and code metrics.

The results are not downloaded automatically to your desktop.

☐ Off (default)

Analysis results are stored locally.

Dependencies

The option to upload to Polyspace Metrics is available only if you select `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`.

If you perform a local analysis on your desktop, you can later upload your results to Polyspace Metrics. Right-click your results file and select **Upload to Metrics**.

Command-Line Information

Parameter: `-add-to-results-repository`
Default: Off
Example (Bug Finder): `polyspace-bug-finder -batch -scheduler NodeHost -add-to-results-repository -password passwordName`
Example (Code Prover): `polyspace-code-prover -batch -scheduler NodeHost -add-to-results-repository -password passwordName`

The password is optional.

The upload uses the Polyspace Metrics server that you set up in the Polyspace user interface. See "Set Up Polyspace Metrics". If you want to explicitly specify the Polyspace Metrics server during upload, use the option `-polyspace-metrics-server serverName:portNumber`. For instance:

```
-add-to-results-repository -polyspace-metrics-server localhost:12427
```

See Also

`Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`

Topics

"Set Up Polyspace Metrics"
"Generate Code Quality Metrics with Polyspace Metrics"

Use fast analysis mode for Bug Finder (`-fast-analysis`)

Run analysis using faster local mode

Description

This option affects a Bug Finder analysis only.

Run analysis using faster local mode. The first run analyzes all files, but subsequent runs analyze only the files that you edited since the previous analysis.

Fast analysis mode is a faster way to analyze code for localized defects and coding rules. When you launch fast analysis, Bug Finder analyzes your code for a subset of defects and coding rules.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Run Settings** node.

Command line: Use the option `-fast-analysis`. See “Command-Line Information” on page 1-312.

Why Use This Option

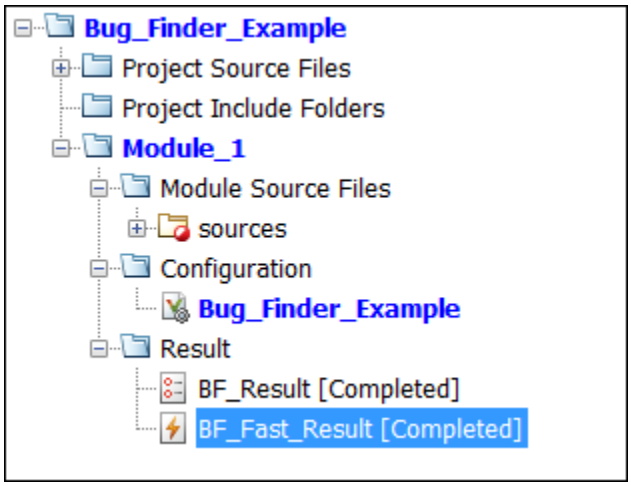
If you use this option, you have to wait less for analysis results from your second analysis onwards. During development, you can frequently run analysis in fast mode and quickly check for new defects or coding rule violations.

Polyspace produces results quickly because the analysis is localized. When you rerun in fast-analysis mode, Polyspace reanalyzes only those files that need to be reanalyzed, regenerating results even more quickly. These situations trigger a reanalysis.

Situation	What Is Reanalyzed
You modified a source file.	Modified source file
You modified a header file.	Source files that include the modified header file (directly or indirectly)
You added or removed an analysis option.	All files
Previous fast-analysis results were not found. For instance, you deleted the results folder.	All files
You upgraded to a later release of Polyspace and ran the analysis.	All files Comments from the previous analysis are retained and imported to the current analysis.

For example, consider a Polyspace project with three `.c` files and you fix a bug in one of the files. When you rerun the analysis, Polyspace reanalyzes only the one file that you changed.

The results of fast analysis appear in a folder separate from the results of normal analysis.



Settings

Default: ☐ Off

☑ On

Polyspace Bug Finder runs in fast-analysis mode. Polyspace analyzes code for only a subset of defects and coding rules. If you have selected any defects or coding rules that are not supported by fast-analysis, you code is not checked for those results.

☐ Off

Polyspace Bug Finder runs in the normal mode. Analysis checks for all selected defects, coding rules, and code metrics.

Tips

Comments Import

If you enter comments in your results, the comments are automatically imported to the next analysis in fast mode.

To import the comments from fast mode results to results of a regular Bug Finder analysis, do one of the following:

- Select **Tools** > **Import Comments**. Navigate to the sibling results folder `BF_Fast_Result` and import comments from the fast mode results.
- When reviewing results of fast mode, enter the comments directly into your code. If you run a regular analysis on this code, the comments are imported to your analysis results.

  For details on how to enter code comments, see “Annotate Code and Hide Known or Acceptable Results”.

Fast Analysis Limitations

In fast analysis mode, you cannot perform these actions:

- You cannot create a new results folder for each run. Even if you choose to create a new result folder, each new run overwrites the previous one.
- You cannot specify constraints using the option `Constraint setup (-data-range-specifications)`.
- You cannot run the analysis on a remote cluster.

Command-Line Information

Parameter: `-fast-analysis`
Default: Off
Example (Bug Finder): `polyspace-bug-finder -sources filename -fast-analysis`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources filename -fast-analysis`

See Also

Topics
"Bug Finder Results Found in Fast Analysis Mode"

Command/script to apply after the end of the code verification (-post-analysis-command)

Specify command or script to be executed after analysis

Description

Specify a command or script to be executed after the analysis.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Advanced Settings** node.

Command line: Use the option `-post-analysis-command`. See “Command-Line Information” on page 1-314.

Why Use This Option

Create scripts for tasks that you want performed after the Polyspace analysis.

For instance, you want to be notified by email that the Polyspace analysis is over. Create a script that sends an email and use this option to execute the script after the Polyspace analysis.

Settings

No Default

Enter full path to the command or script, or click  to navigate to the location of the command or script. After the analysis, this script is executed.

The script is executed in the Polyspace results folder. In your script, consider the results folder as the current folder for relative paths to other files.

For a Perl script, in Windows, specify the full path to the Perl executable followed by the full path to the script. For example, to specify a Perl script `send_email.pl` that sends an email once the analysis is over, enter *`polyspaceroot`*`\sys\perl\win32\bin\perl.exe <`*`absolute_path`*`> \send_email.pl`. Here, *`polyspaceroot`* is the location of the current Polyspace installation, such as `C:\Program Files\Polyspace\R2019a\`, and *`<absolute_path>`* is the location of the Perl script.

Tips

Running post analysis commands on the server

If you perform verification on a remote server, after verification, the software executes your command on the server, not on the client desktop. If your command executes a script, the script must be present on the server.

For instance, if you specify the command, `/local/utils/send_mail.sh`, the Shell script `send_email.sh` must be present on the server in `/local/utils/`. The software does not copy the

script `send_email.sh` from your desktop to the server before executing the command. If the script is not present on the server, you encounter an error. Sometimes, there are multiple servers that the MATLAB Job Scheduler can run the verification on. Place the script on each of the servers because you do not control which server eventually runs your verification.

Running post analysis commands in the Polyspace user interface

To test the use of this option, run the following Perl script from a folder containing a Polyspace project (`.psprj` file). The script parses the latest Polyspace log file in the folder `Module_1\CP_Result` and writes the current project name and date to a file `report.txt`. The file is saved in `Module_1\CP_Result`.

```
foreach my $file (`ls Module_1\\CP_Result\\Polyspace_*.log`) {
  open (FH, $file);

while ($line = <FH>) {
    if ($line =~ m/Ending at: (.*)/) {
      $date=$1;
    }
    if ($line =~ m/-prog=(.*)/) {
      $project=$1;
    }
  }
}

my $filename = 'report.txt';
open(my $fh, '>', $filename) or die "Could not open file '$filename' $!";

print $fh "date=$date\n";
print $fh "project=$project\n";

close $fh;
```

In Linux, you can specify the Perl script for this option.

In Windows, instead of specifying the Perl script directly, specify a `.bat` file that invokes Perl and runs this script. For instance, the `.bat` file can contain the following line (assuming that the `.bat` file and `.pl` file are in the Polyspace project folder). Depending on your MATLAB installation, change the path to `perl.exe` appropriately.

```
"C:\Program Files\MATLAB\R2018b\sys\perl\win32\bin\perl.exe" command.pl
```

Run Code Prover. Check that the folder `Module_1\CP_Result` contains the file `report.txt` with the project name and date.

Command-Line Information

Parameter: `-post-analysis-command`
Value: Path to executable file or command in quotes
No Default
Example in Linux (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-post-analysis-command \`pwd\`/send_email.pl`
Example in Linux (Code Prover) : `polyspace-code-prover -sources` *`file_name`* `-post-analysis-command \`pwd\`/send_email.pl`

Example in Linux (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-post-analysis-command` `` `pwd`/send_email.pl ``
Example in Linux (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-post-analysis-command` `` `pwd`/send_email.pl ``
Example in Windows: `polyspace-bug-finder -sources` *`file_name`* `-post-analysis-command "C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\send_email"`

Note that in Windows, you use the full path to the Perl executable.

See Also

`Command/script to apply to preprocessed files (-post-preprocessing-command)`

Topics
“Specify Polyspace Analysis Options”

Automatic Orange Tester (`-automatic-orange-tester`)

Specify that Automatic Orange Tester must be executed after verification

Description

This option affects a Code Prover analysis only. Use this option only if you review the Code Prover results in the Polyspace desktop products.

Specify that the Automatic Orange Tester must be executed at the end of the verification.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Advanced Settings** node. See “Dependency” on page 1-316 for other options you must also enable.

Command line: Use the option `-automatic-orange-tester`. See “Command-Line Information” on page 1-317.

Why Use This Option

The Automatic Orange Tester runs dynamic tests on your code. The dynamic tests help you determine if an orange check represents a real run-time error or an imprecision of Polyspace analysis. For a tutorial, see “Test Orange Checks for Run-Time Errors” (Polyspace Code Prover).

To run the Automatic Orange Tester after verification, you must select this option *before verification*. During verification, Polyspace generates additional source code to test each orange check for errors. When you run the Automatic Orange Tester later, the software uses this instrumented code for testing.

Settings

☑ On

After verification, when you run the Automatic Orange Tester, Polyspace creates tests for unproven code and runs them.

☐ Off (default)

You cannot launch the Automatic Orange Tester after verification.

Dependency

This option is available only if you set `Source code language (-lang)` to `C` or `C-CPP`.

Tips

- To launch the Automatic Orange Tester, after verification, open your results. Select **Tools** > **Automatic Orange Tester**.

- When using the automatic orange tester, you cannot:
  - Select **Division round down** under **Target & Compiler**.
  - Select the options `c18`, `tms320c3c`. `x86_64` or `sharc21x61` for **Target & Compiler** > **Target processor type**.
  - Specify the type `char` as 16-bit or `short` as 8-bit using the option `mcpu...(Advanced)` for **Target & Compiler** > **Target processor type**. For the same option, you must specify the type `pointer` as 32-bit.
  - Specify global asserts in the code, having the form `Pst_Global_Assert(A,B)`. In global assert mode, you cannot use **Constraint setup** under **Inputs & Stubbing**.
  - Select these options related to floating-point verification: **Subnormal detection mode** and **Consider non finite floats**.

Command-Line Information

Parameter: `-automatic-orange-tester`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-lang c -automatic-orange-tester`

See Also

`Maximum loop iterations (-automatic-orange-tester-loop-max-iteration)` | `Maximum test time (-automatic-orange-tester-timeout)` | `Number of automatic tests (-automatic-orange-tester-tests-number)`

Topics

“Specify Polyspace Analysis Options”
“Test Orange Checks for Run-Time Errors” (Polyspace Code Prover)
“Limitations of Automatic Orange Tester” (Polyspace Code Prover)

Maximum loop iterations (-automatic-orange-tester-loop-max-iteration)

Specify number of loop iterations after which Automatic Orange Tester considers infinite loop

Description

This option affects a Code Prover analysis only. Use this option only if you review the Code Prover results in the Polyspace desktop products.

Specify number of loop iterations after which the Automatic Orange Tester considers the loop to be infinite. Specifying a large number decreases the possibility of identifying an infinite loop incorrectly, but takes more time to complete.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Advanced Settings** node. See “Dependencies” on page 1-318 for other options you must also enable.

Command line: Use the option `-automatic-orange-tester-loop-max-iteration`. See “Command-Line Information” on page 1-318.

Settings

Default: 1000

Enter number of loop iterations. The maximum value that the software supports is 1000.

Dependencies

This option is enabled only if you set the following options:

- Set `Source code language (-lang)` to `C` or `C-CPP`.
- Specify the option `Automatic Orange Tester (-automatic-orange-tester)`.

Command-Line Information

Parameter: `-automatic-orange-tester-loop-max-iteration`
Value: *`positive integer`*
Default: 1000
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-lang c -automatic-orange-tester -automatic-orange-tester-loop-max-iteration 500`

See Also

`Automatic Orange Tester (-automatic-orange-tester)`

Topics

“Specify Polyspace Analysis Options”
“Test Orange Checks for Run-Time Errors” (Polyspace Code Prover)

Number of automatic tests (-automatic-orange-tester-tests-number)

Specify number of tests that Automatic Orange Tester must run

Description

This option affects a Code Prover analysis only. Use this option only if you review the Code Prover results in the Polyspace desktop products.

Specify number of tests that you want the Automatic Orange Tester to run. The more the number of tests, the greater the possibility of finding a run-time error, but longer it takes to complete.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Advanced Settings** node. See “Dependencies” on page 1-319 for other options you must also enable.

Command line: Use the option `-automatic-orange-tester-tests-number`. See “Command-Line Information” on page 1-319.

Settings

Default: 500

Enter number of tests up to a maximum of 100,000.

Dependencies

This option is enabled only if you set the following options:

- Set `Source code language (-lang)` to `C` or `C-CPP`.
- Specify the option `Automatic Orange Tester (-automatic-orange-tester)`.

Command-Line Information

Parameter: `-automatic-orange-tester-tests-number`
Value: *`positive integer`*
Default: 500
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-lang c -automatic-orange-tester -automatic-orange-tester-tests-number 500`

See Also

`Automatic Orange Tester (-automatic-orange-tester)`

Topics

“Specify Polyspace Analysis Options”
“Test Orange Checks for Run-Time Errors” (Polyspace Code Prover)

Maximum test time (-automatic-orange-tester-timeout)

Specify time in seconds allowed for a single test in Automatic Orange Tester

Description

This option affects a Code Prover analysis only. Use this option only if you review the Code Prover results in the Polyspace desktop products.

Specify time in seconds allowed for a single test. After this time is over, the Automatic Orange Tester proceeds to the next test. Increasing this time reduces number of tests that do not complete, but increases total verification time.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Advanced Settings** node. See “Dependencies” on page 1-320 for other options you must also enable.

Command line: Use the option `-automatic-orange-tester-timeout`. See “Command-Line Information” on page 1-320.

Settings

Default: 5

Enter time in seconds. The maximum value that the software supports is 60.

Dependencies

This option is enabled only if you set the following options:

- Set `Source code language (-lang)` to `C` or `C-CPP`.
- Specify the option `Automatic Orange Tester (-automatic-orange-tester)`.

Command-Line Information

Parameter: `-automatic-orange-tester-timeout`
Value: *`time`*
Default: 5
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-lang c -automatic-orange-tester -automatic-orange-tester-test-timeout 10`

See Also

`Automatic Orange Tester (-automatic-orange-tester)`

Topics

“Specify Polyspace Analysis Options”
“Test Orange Checks for Run-Time Errors” (Polyspace Code Prover)

Other

Specify additional flags for analysis

Description

This option is useful only if you run an analysis in the user interface of the Polyspace desktop products.

Enter command-line-style flags such as `-max-processes`.

Set Option

In your project configuration, the option is on the **Advanced Settings** node. You can enter multiple options in this field. If you enter the same option multiple times with different arguments, the analysis uses your last argument.

Why Use This Option

Use this option to add nonofficial or command-line only options to the analyzer.

If you have to add several command line options, you can save them in a text file and specify the file using the option `-options-file`. You can reuse the options file across projects.

Tip

Nonofficial options: In rare circumstances, to work around very specific issues, MathWorks Technical Support might provide you some undocumented options. If you are running verification from the user interface, you use the **Other** field in the **Configuration** pane to enter the options. Sometimes, the options and their arguments have to be preceded by extra flags. When providing you the option, Technical Support will let you know if the extra flags are required.
Possible Flags: `-extra-flags | -c-extra-flags | -cpp-extra-flags | -cfe-extra-flags | -il-extra-flags`
Example (Bug Finder): `polyspace-bug-finder -extra-flags -`*`option-name`* `-extra-flags` *`option_param`*
Example (Code Prover): `polyspace-code-prover -extra-flags -`*`option-name`* `-extra-flags` *`option_param`*
Example (Bug Finder Server): `polyspace-bug-finder-server -extra-flags -`*`option-name`* `-extra-flags` *`option_param`*
Example (Code Prover Server): `polyspace-code-prover-server -extra-flags -`*`option-name`* `-extra-flags` *`option_param`*

Analysis Options, Command-Line Only

-asm-begin -asm-end

Exclude compiler-specific `asm` functions from analysis

Syntax

`-asm-begin "`*`mark1[,mark2,...]`*`" -asm-end "`*`mark1[,mark2,...]`*`"`

Description

`-asm-begin "`*`mark1[,mark2,...]`*`" -asm-end "`*`mark1[,mark2,...]`*`"` excludes compiler-specific assembly language source code functions from the analysis. You must use these two options together.

Polyspace recognizes most inline assemblers by default. Use the option only if compilation errors occur due to introduction of assembly code. For more information, see "Assembly Code" (Polyspace Code Prover).

Mark the offending code block by two `#pragma` directives, one at the beginning of the assembly code and one at the end. In the command usage, give these marks in the same order for `-asm-begin` as they are for `-asm-end`.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

A block of code is delimited by `#pragma start1` and `#pragma end1`. These names must be in the same order for their respective options. Either:

```
-asm-begin "start1" -asm-end "end1"
```

or

```
-asm-begin "mark1,...markN,start1" -asm-end "mark1,...markN,end1"
```

The following example marks two functions for exclusion, `foo_1` and `foo_2`.

Code:

```
#pragma asm_begin_foo
int foo(void) { /* asm code to be ignored by Polyspace */ }
#pragma asm_end_foo

#pragma asm_begin_bar
void bar(void) { /* asm code to be ignored by Polyspace */ }
#pragma asm_end_bar
```

Polyspace Command:

- Bug Finder:

```
polyspace-bug-finder -lang c -asm-begin "asm_begin_foo,asm_begin_bar"
          -asm-end "asm_end_foo,asm_end_bar"
```

- Code Prover:

```
polyspace-code-prover -lang c -asm-begin "asm_begin_foo,asm_begin_bar"
         -asm-end "asm_end_foo,asm_end_bar"
```

- Bug Finder Server:

```
polyspace-bug-finder-server -lang c -asm-begin "asm_begin_foo,asm_begin_bar"
          -asm-end "asm_end_foo,asm_end_bar"
```

- Code Prover Server:

```
polyspace-code-prover-server -lang c -asm-begin "asm_begin_foo,asm_begin_bar"
         -asm-end "asm_end_foo,asm_end_bar"
```

`asm_begin_foo` and `asm_begin_bar` mark the beginning of the assembly source code sections to be ignored. `asm_end_foo` and `asm_end_bar` mark the end of those respective sections.

See Also

Topics
“Specify Polyspace Analysis Options”

-author

Specify project author

Syntax

`-author "value"`

Description

`-author "value"` assigns an author to the Polyspace project. The name appears as the project owner in Polyspace Metrics and on generated reports.

The default value is the user name of the current user, given by the DOS or UNIX command `whoami`.

In the user interface of the Polyspace desktop products, select  to specify the Project name, Version, and Author parameters in the **Polyspace Project – Properties** dialog box.

Examples

Assign a project author to your Polyspace Project.

- Bug Finder:

  ```
  polyspace-bug-finder -author "John Smith"
  ```
- Code Prover:

  ```
  polyspace-code-prover -author "John Smith"
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -author "John Smith"
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -author "John Smith"
  ```

See Also

`-date` | `-prog`

Topics

“Specify Polyspace Analysis Options”

-code-behavior-specifications

Map imprecisely analyzed function to standard function for precise analysis

Syntax

`-code-behavior-specifications` *`file1`*`[, `*`file2`*`, [...]]`

Description

`-code-behavior-specifications` *`file1`*`[, `*`file2`*`, [...]]` specifies XML files that allow you to associate behaviors with elements of your code. For instance, you can:

- Map your library functions to corresponding standard functions that Polyspace recognizes. Mapping to standard library functions can help with precision improvement or automatic detection of new threads.
- Specify a function as forbidden.

If you run verification from the command line, specify the absolute path to the XML files or path relative to the folder from which you run the command. If you run verification from the user interface (desktop products only), specify the option along with an absolute path to the XML file in the **Other** field. See `Other`.

A sample template file `code-behavior-specifications-sample.xml` shows the XML syntax. The file is in *`polyspaceroot`*`\polyspace\verifier\cxx\` where *`polyspaceroot`* is the Polyspace installation folder.

Using Option for Precision Improvement

XML Syntax: `<function name="`*`custom_function`*`" std="`*`std_function`*`"> </function>`

Use this entry in the XML file to reduce the number of orange checks from imprecise Code Prover analysis of your function (or false negatives from an imprecise Bug Finder analysis). Sometimes, the verification does not analyze certain kinds of functions precisely because of inherent limitations in static verification. In those cases, if you find a standard function that is a close analog of your function, use this mapping. Though your function itself is not analyzed, the analysis is more precise at the locations where you call the function. For instance, if the verification cannot analyze your function `cos32` precisely and considers full range for its return value, map it to the `cos` function for a return value in [-1,1].

The verification ignores the body of your function. However, the verification emulates your function behavior in the following ways:

- The verification assumes the same return values for your function as the standard function.

  For instance, if you map your function `cos32` to the standard function `cos`, the verification assumes that `cos32` returns values in [-1,1].
- The verification checks for the same issues as it checks with the standard function.

  For instance, if you map your function `acos32` to the standard function `acos`, the `Invalid use of standard library routine` check determines if the argument of `acos32` is in [-1,1].

The functions that you can map to include:

- Standard library functions from `math.h`.
- Memory management functions from `string.h`.
- `__ps_meminit`: A function specific to Polyspace that initializes a memory area.

  Sometimes, the verification does not recognize your memory initialization function and produces an orange `Non-initialized local variable` check on a variable that you initialized through this function. If you know that your memory initialization function initializes the variable through its address, map your function to `__ps_meminit`. The check turns green.
- `__ps_lookup_table_clip`: A function specific to Polyspace that returns a value within the range of the input array.

  Sometimes, the verification considers full range for the return values of functions that look up values in large arrays (look-up table functions). If you know that the return value of a look-up table function must be within the range of values in its input array, map the function to `__ps_lookup_table_clip`.

  In code generated from models, the verification by default makes this assumption for look-up table functions. To identify if the look-up table uses linear interpolation and no extrapolation, the verification uses the function names. Use the mapping only for handwritten functions, for instance, functions in a C/C++ S-Function block. The names of those functions do not follow specific conventions. You must explicitly specify them.

See also “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries”.

Using Option for Concurrency Detection

XML Syntax: `<function name="`*`custom_function`*`" std="`*`std_function`*`"> </function>`

Use this entry in the XML file for automatic detection of thread-creation functions and functions that begin and end critical sections. Polyspace supports automatic detection for certain families of multitasking primitives only. Extend the support using this XML entry.

If your thread-creation function, for instance, does not belong to one of the supported families, map your function to a supported concurrency primitive.

See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”.

Using Option for Blacklisting Functions

This section applies only to a Bug Finder analysis.

XML Syntax: `<function name="`*`function_name`*`" behavior="FORBIDDEN_FUNC"> </function>`

Use this entry in the XML file to specify a list of functions that you want to prohibit from your source code.

See “Flag Deprecated or Unsafe Functions Using Bug Finder Checkers”.

Examples

The examples in the next sections refer to a Code Prover analysis. For Bug Finder examples, see:

- “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries”
- “Flag Deprecated or Unsafe Functions Using Bug Finder Checkers”
- “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”

Specify Mapping to Standard Function

You can adapt the sample mapping XML file provided with your Polyspace installation and map your function to a standard function.

Suppose the default verification produces an orange `User assertion` check on this code:

```
double x = acos32(1.0) ;
assert(x <= 2.0);
```

Suppose you know that the function `acos32` behaves like the function `acos` and the return value is 0. You expect the check on the `assert` statement to be green. However, the verification considers that `acos32` returns any value in the range of type `double` because `acos32` is not precisely analyzed. The check is orange. To map your function `acos32` to `acos`:

1. Copy the file `code-behavior-specifications-sample.xml` from *`polyspaceroot`* `\polyspace\verifier\cxx\` to another location, for instance, `"C:\Polyspace_projects\Common\Config_files"`. Change the write permissions on the file.
2. To map your function to a standard function, modify the contents of the XML file. To map your function `acos32` to the standard library function `acos`, change the following code:

   ```
   <function name="my_lib_cos" std="acos"> </function>
   ```

   To:

   ```
   <function name="acos32" std="acos"> </function>
   ```
3. Specify the location of the file for verification:

   - Code Prover:

     ```
     polyspace-code-prover -code-behavior-specifications
       "C:\Polyspace_projects\Common\Config_files
        \code-behavior-specifications-sample.xml"
     ```
   - Code Prover Server:

     ```
     polyspace-code-prover-server -code-behavior-specifications
       "C:\Polyspace_projects\Common\Config_files
        \code-behavior-specifications-sample.xml"
     ```

Specify Mapping to Standard Function with Argument Remapping

Sometimes, the arguments of your function do not map one-to-one with arguments of the standard function. In those cases, remap your function argument to the standard function argument. For instance:

- `__ps_lookup_table_clip`:

  This function specific to Polyspace takes only a look-up table array as argument and returns values within the range of the look-up table. Your look-up table function might have additional arguments besides the look-up table array itself. In this case, use argument remapping to specify which argument of your function is the look-up table array.

For instance, suppose a function `my_lookup_table` has the following declaration:

```
double my_lookup_table(double u0, const real_T *table,
                                       const double *bp0);
```

The second argument of your function `my_lookup_table` is the look-up table array. In the file `code-behavior-specifications-sample.xml`, add this code:

```
<function name="my_lookup_table" std="__ps_lookup_table_clip">
    <mapping std_arg="1" arg="2"></mapping>
</function>
```

When you call the function:

```
res = my_lookup_table(u, table10, bp);
```

The verification interprets the call as:

```
res =__ps_lookup_table_clip(table10);
```

The verification assumes that the value of `res` lies within the range of values in `table10`.

- `__ps_meminit`:

  This function specific to Polyspace takes a memory address as the first argument and a number of bytes as the second argument. The function assumes that the bytes in memory starting from the memory address are initialized with a valid value. Your memory initialization function might have additional arguments. In this case, use argument remapping to specify which argument of your function is the starting address and which argument is the number of bytes.

  For instance, suppose a function `my_meminit` has the following declaration:

  ```
  void my_meminit(enum InitKind k, void* dest, int is_aligned,
                                unsigned int size);
  ```

  The second argument of your function is the starting address and the fourth argument is the number of bytes. In the file `code-behavior-specifications-sample.xml`, add this code:

  ```
  <function name="my_meminit" std="__ps_meminit">
      <mapping std_arg="1" arg="2"></mapping>
      <mapping std_arg="2" arg="4"></mapping>
  </function>
  ```

  When you call the function:

  ```
  my_meminit(INIT_START_BY_END, &buffer, 0, sizeof(buffer));
  ```

  The verification interprets the call as:

  ```
  __ps_meminit(&buffer, sizeof(buffer));
  ```

  The verification assumes that `sizeof(buffer)` number of bytes starting from `&buffer` are initialized.

- `memset`: Variable number of arguments.

  If your function has variable number of arguments, you cannot map it directly to a standard function without explicit argument remapping. For instance, say your function is declared as:

  ```
  void* my_memset(void*, int, size_t, ...)
  ```

To map the function to the `memset` function, use the following mapping:

```
<function name="my_memset" std="memset">
    <mapping std_arg="1" arg="1"></mapping>
    <mapping std_arg="2" arg="2"></mapping>
    <mapping std_arg="3" arg="3"></mapping>
</function>
```

Effect of Mapping on Precision

These examples show the result of mapping certain functions to standard functions:

- `my_acos` → `acos`:

  If you use the mapping, the `User assertion` check turns green. The verification assumes that the return value of `my_acos` is 0.

  - *Before mapping*:

    ```
    double x = my_acos(1.0);
    assert(x <= 2.0);
    ```

  - *Mapping specification*:

    ```
    <function name="my_acos" std="acos">
    </function>
    ```

  - *After mapping*:

    ```
    double x = my_acos(1.0);
    assert(x <= 2.0);
    ```

- `my_sqrt` → `sqrt`:

  If you use the mapping, the `Invalid use of standard library routine` check turns red. Otherwise, the verification does not check whether the argument of `my_sqrt` is nonnegative.

  - *Before mapping*:

    ```
    res = my_sqrt(-1.0);
    ```

  - *Mapping specification*:

    ```
    <function name="my_sqrt" std="sqrt">
    </function>
    ```

  - *After mapping*:

    ```
    res = my_sqrt(-1.0);
    ```

- `my_lookup_table` (argument 2) → `__ps_lookup_table_clip` (argument 1):

  If you use the mapping, the `User assertion` check turns green. The verification assumes that the return value of `my_lookup_table` is within the range of the look-up table array `table`.

  - *Before mapping*:

    ```
    double table[3] = {1.1, 2.2, 3.3}
    .
    .
    double res = my_lookup_table(u, table, bp);
    assert(res >= 1.1 && res <= 3.3);
    ```

- *Mapping specification*:

```
<function name="my_lookup_table" std="__ps_lookup_table_clip">
    <mapping std_arg="1" arg="2"></mapping>
</function>
```

- *After mapping*:

```
double table[3] = {1.1, 2.2, 3.3}
.
.
res_real = my_lookup_table(u, table9, bp);
assert(res_real >= 1.1 && res_real <= 3.3);
```

- `my_meminit` → `__ps_meminit`:

  If you use the mapping, the `Non-initialized local variable` check turns green. The verification assumes that all fields of the structure `x` are initialized with valid values.

  - *Before mapping*:

```
struct X {
  int field1 ;
  int field2 ;
};
.
.
struct X x;
my_meminit(&x, sizeof(struct X));
return x.field1;
```

  - *Mapping specification*:

```
<function name="my_meminit" std="__ps_meminit">
     <mapping std_arg="1" arg="1"></mapping>
     <mapping std_arg="2" arg="2"></mapping>
</function>
```

  - *After mapping*:

```
struct X {
  int field1 ;
  int field2 ;
};
.
.
struct X x;
my_meminit(&x, sizeof(struct X));
return x.field1;
```

- `my_meminit` → `__ps_meminit`:

  If you use the mapping, the `Non-initialized local variable` check turns red. The verification assumes that only the field `field1` of the structure `x` is initialized with valid values.

  - *Before mapping*:

```
struct X {
  int field1 ;
  int field2 ;
};
```

```
.
.
struct X x;
my_meminit(&x, sizeof(int));
return x.field2;
```

- *Mapping specification*:

  ```
  <function name="my_meminit" std="__ps_meminit">
  </function>
  ```

- *After mapping*:

  ```
  struct X {
    int field1 ;
    int field2 ;
  };
  .
  .
  struct X x;
  my_meminit(&x, sizeof(int));
  return x.field2;
  ```

See Also

Topics
“Specify Polyspace Analysis Options”

Introduced in R2016b

-custom-target

Create a custom target processor with specific data type sizes

Syntax

`-custom-target` *`target_sizes`*

Description

`-custom-target` *`target_sizes`* defines a custom target processor for the Polyspace analysis. The target processor definition includes sizes in bytes of fundamental data types, signedness of plain `char`, alignment of structures and underlying types of standard `typedef`-s such as `size_t`, `ptrdiff_t` and `wchar_t`.

`target_sizes` is a comma-separated list specifying these values. From left to right, the values are the following. If a data type is not supported, -1 is used for its size.

Specification	Possible Values
Whether plain `char` is signed	`true` or `false`
Size of `char` in bits Other sizes are in bytes.	Number
Size of `short`	Number
Size of `int`	Number
Size of `short long`	Number
Size of `long`	Number
Size of `long long`	Number
Size of `float`	Number
Size of `double`	Number
Size of `long double`	Number
Size of pointer	Number
Maximum alignment of all integer types	Number
Maximum alignment of variables of type `struct` or `union`	Number
Endianness	`little` or `big`
Underlying type of `size_t`	`unknown`, `unsigned_char`, `unsigned_short`, `unsigned_int`, `unsigned_long`, or `unsigned_long_long`
Underlying type of `ptrdiff_t`	`unknown`, `signed_char`, `short`, `int`, `long`, or `long_long`
Underlying type of `wchar_t`	`unknown`, `short`, `unsigned_short`, `int`, `unsigned_int`, `long`, or `unsigned_long`

Typically, this option is used when the `polyspace-configure` command creates an options file for the subsequent Polyspace analysis. However, you can directly enter this option when manually writing options files. This option is useful in situations where your target specifications are not covered by one of the predefined target processors. See `Target processor type (-target)`.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

An usage of the option looks like this:

```
-custom-target false,8,2,4,-1,4,8,4,8,8,4,8,1,little,unsigned_int,int,unsigned_int
```

The option argument translates to the following target specification.

Specification	Possible Values
Whether plain `char` is signed	`false`
Size of `char`	8 bits
Size of `short`	2 bytes
Size of `int`	4 bytes
Size of `short long`	`short long` is not supported.
Size of `long`	4 bytes
Size of `long long`	8 bytes
Size of `float`	4 bytes
Size of `double`	8 bytes
Size of `long double`	8 bytes
Size of pointer	4 bytes
Maximum alignment of all integer types	8 bytes
Maximum alignment of variables of type `struct` or `union`	1 byte
Endianness	`little`
Underlying type of `size_t`	`unsigned_int`
Underlying type of `ptrdiff_t`	`int`
Underlying type of `wchar_t`	`unsigned_int`

See Also

`Generic target options` | `Target processor type (-target)`

Topics

“Specify Polyspace Analysis Options”

-date

Specify date of analysis

Syntax

`-date "`*`date`*`"`

Description

`-date "`*`date`*`"` specifies the date stamp for the analysis in the format `dd/mm/yyyy`. By default the value is the date the analysis starts.

Examples

Assign a date to your Polyspace Project:

- Bug Finder:

  ```
  polyspace-bug-finder -date "15/03/2012"
  ```
- Code Prover:

  ```
  polyspace-code-prover -date "15/03/2012"
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -date "15/03/2012"
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -date "15/03/2012"
  ```

See Also

`-author` | `-date`

Topics

“Specify Polyspace Analysis Options”

-doc | -documentation

Display Polyspace documentation in help browser

Syntax

```
-doc
-documentation
```

Description

`-doc` and `-documentation` opens Polyspace documentation in a help browser. You can see information such as getting started, workflows and reference pages for commands and analysis options. You can also search through the documentation in the help browser.

Examples

Display Polyspace documentation in a help browser:

- Bug Finder:

  ```
  polyspace-bug-finder -doc
  polyspace-bug-finder -documentation
  ```

- Code Prover:

  ```
  polyspace-code-prover -doc
  polyspace-code-prover -documentation
  ```

- Bug Finder Server:

  ```
  polyspace-bug-finder-server -doc
  polyspace-bug-finder-server -documentation
  ```

- Code Prover Server:

  ```
  polyspace-code-prover-server -doc
  polyspace-code-prover-server -documentation
  ```

See Also

`-h[elp]`

-dump-preprocessing-info

Show all macros implicitly defined during a particular analysis

Syntax

```
-dump-preprocessing-info
```

Description

`-dump-preprocessing-info` prints all the macros implicitly defined (or undefined) during a particular Polyspace analysis. The macro definitions come from:

- Your specification for the option `Compiler (-compiler)`

  Polyspace emulates a compiler by defining the compiler-specific macros.
- Macros defined (or undefined) in the Polyspace implementation of Standard Library headers
- Macros that you explicitly define (or undefine) using the options `Preprocessor definitions (-D)` and `Disabled preprocessor definitions (-U)`

Use this option only if you want to know how Polyspace defines a specific macro. In case you want to use a different definition for the macro, you can then override the current definition.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`. On the **Output Summary** pane, you can see each macro definition on a separate line. You can search for the macro name in the user interface and click the line with the macro name to see further details in the **Detail** pane.

Examples

Suppose that you use the ARM v6 compiler for building your source code. For the Polyspace analysis, you use the value `armclang` for the option `Compiler (-compiler)`. Suppose that you want to know what Polyspace uses as definition for the macro `__ARM_ARCH`.

1. Enter the following command and pipe the console output to a file that you can search later:

   ```
   polyspace-bug-finder -sources aFile.c -compiler armclang -dump-preprocessing-info
   ```

   `aFile.c` can be a simple C file. You can also replace `polyspace-bug-finder` with `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server`.
2. Search for `__ARM_ARCH` in the file containing the console output. You can see the line with the macro definition:

   ```
   Remark: Definition of macro __ARM_ARCH (pre-processing __polyspace__stdstubs.c)
   |#define __ARM_ARCH 8
   |defined by syntax extension xml file
   |predefined macro
   ```

In this example, the macro is set to the value 8.

- To override this macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine this macro, use the option `Disabled preprocessor definitions (-U)`.

See Also

`Compiler (-compiler)`

Topics

“Specify Polyspace Analysis Options”

-generate-launching-script-for

Extract information from project file

Syntax

`-generate-launching-script-for` *`PRJFILE`*

Description

`-generate-launching-script-for` *`PRJFILE`* extracts information from a project file *`PRJFILE`* (created in the user interface of the Polyspace desktop products) so that you can run an analysis from the command line. For each project module and each configuration in each module, a folder is created containing the following files::

- `source_command.txt` — List of source files for the `-sources-list-file` option.
- `options_command.txt` — List of the analysis options for the `-options-file` option.
- `temporal_exclusions.txt` — List of temporal exclusions, generated only if you specify the `Temporally exclusive tasks (-temporal-exclusions-file)` option.
- `.polyspace_conf.psprj` — A copy of the project file Polyspace used to generate the scripting files.
- `launchingCommand.sh` (UNIX) or `launchingCommand.bat` (DOS) — shell script that calls the correct commands. The script also calls any options that cannot be given to the `-options-file` command, such as `-batch` or `-add-to-results-repository`. You can give this file additional analysis options as parameters.

Note The script that Polyspace generates runs the same analysis that Polyspace runs from the user interface. If your project runs in the Polyspace user interface, the script will run from the command line.

Examples

Extract information to run `myproject` from the command line. Use this option with the desktop binary `polyspace`:

- Bug Finder:

  ```
  polyspace -generate-launching-script-for myproject.psprj -bug-finder
  ```

- Code Prover:

  ```
  polyspace -generate-launching-script-for myproject.psprj
  ```

See Also

Topics
“Configure Polyspace Analysis Options in User Interface and Generate Scripts”

-h | -help

Display list of possible options

Syntax

```
-h
-help
```

Description

`-h` and `-help` display the list of possible options in the command window along with option argument syntax.

Examples

Display the command-line help:

- Bug Finder:

  ```
  polyspace-bug-finder -h
  polyspace-bug-finder -help
  ```

- Code Prover:

  ```
  polyspace-code-prover -h
  polyspace-code-prover -help
  ```

- Bug Finder Server:

  ```
  polyspace-bug-finder-server -h
  polyspace-bug-finder-server -help
  ```

- Code Prover Server:

  ```
  polyspace-code-prover-server -h
  polyspace-code-prover-server -help
  ```

`-doc | -documentation`

-I

Specify include folder for compilation

Syntax

`-I` *`folder`*

Description

`-I` *`folder`* specifies a folder that contains include files required for compiling your sources. You can specify only one folder for each instance of `-I`. However, you can specify this option multiple times.

The analysis looks for include files relative to the folder paths that you specify. For instance, if your code contains the preprocessor directive `#include<../mylib.h>` and you include the folder:

```
C:\My_Project\MySourceFiles\Includes
```

the folder `C:\My_Project\MySourceFiles` must contain a file `mylib.h`.

The analysis automatically includes the `./sources` folder (if it exists) after the include folders that you specify.

Examples

Include two folders with the analysis:

- Bug Finder:

  ```
  polyspace-bug-finder -I /com1/inc -I /com1/sys/inc
  ```

- Code Prover:

  ```
  polyspace-code-prover -I /com1/inc -I /com1/sys/inc
  ```

- Bug Finder Server:

  ```
  polyspace-bug-finder-server -I /com1/inc -I /com1/sys/inc
  ```

- Code Prover Server:

  ```
  polyspace-code-prover-server -I /com1/inc -I /com1/sys/inc
  ```

The source folder is implicitly included. Include files in the source folder can be found automatically without explicit inclusion of the source folder with the `-I` option.

See Also

Topics
“Specify Polyspace Analysis Options”

-import-comments

Import review information from previous analysis

Syntax

`-import-comments` *`resultsFolder`*

Description

`-import-comments` *`resultsFolder`* imports the review information (status, severity and additional notes) from a previous analysis, as specified by the results folder.

You can import review information from the same type of results only. For instance:

- You cannot import review information from a results of a Bug Finder checker to a Code Prover run-time check. Even when the checker names sound similar, the underlying semantics of Bug Finder and Code Prover can be different. The only exception is checkers for coding rules. You can import comments between Bug Finder and Code Prover for coding rule violations.
- You cannot import review information from results of a file-by-file verification in Code Prover to results of a regular Code Prover verification.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

Import review information from the previous results:

- Bug Finder:

  ```
  polyspace-bug-finder -sources filename
         -import-comments C:\Results\myProj\1.2
  ```

- Code Prover:

  ```
  polyspace-code-prover -sources filename
          -import-comments C:\Results\myProj\1.2
  ```

- Bug Finder Server:

  ```
  polyspace-bug-finder-server -sources filename
         -import-comments C:\Results\myProj\1.2
  ```

- Code Prover Server:

  ```
  polyspace-code-prover-server -sources filename
          -import-comments C:\Results\myProj\1.2
  ```

See Also

`-v[ersion]` | `polyspace-comments-import`

Topics
“Import Review Information from Previous Polyspace Analysis”

-list-all-values

Display valid option arguments for a given command-line option

Syntax

`-list-all-values` *`option`*

Description

`-list-all-values` *`option`* displays all the valid option arguments for the command-line option *`option`*.

Examples

Display the valid option arguments for option `-misra3`:

- Polyspace Bug Finder:

  ```
  polyspace-bug-finder -list-all-values -misra3
  ```

- Polyspace Code Prover:

  ```
  polyspace-code-prover -list-all-values -misra3
  ```

- Polyspace Bug Finder Server:

  ```
  polyspace-bug-finder-server -list-all-values -misra3
  ```

- Polyspace Code Prover Server:

  ```
  polyspace-code-prover-server -list-all-values -misra3
  ```

See Also

Topics
“Specify Polyspace Analysis Options”

Introduced in R2020a

-max-processes

Specify maximum number of processors for analysis

Syntax

`-max-processes` *num*

Description

`-max-processes` *num* specifies the maximum number of processes that you want the analysis to use. On a multicore system, the software parallelizes the analysis and creates the specified number of processes to speed up the analysis. The valid range of *num* is 1 to 128.

Unless you specify this option, a Code Prover verification uses up to four processes. If you have fewer than four processes, the verification uses the maximum available number. To increase or restrict the number of processes, use this option.

Unless you specify this option, a Bug Finder analysis uses the maximum number of available processes. Use this option to restrict the number of processes used.

To use this option effectively, determine the number of processors available for use. If the number of processes you create is greater than the number of processors available, the analysis does not benefit from the parallelization. Check the system information in your operating system.

Note that when you start a verification, a message states the number of logical processors detected on your system. However, the analysis is parallelized to the physical processor cores on a machine. Multithreading implementations such as hyper-threading is not taken into account.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

Disable parallel processing during the analysis:

- Bug Finder:

  ```
  polyspace-bug-finder -max-processes 1
  ```
- Code Prover:

  ```
  polyspace-code-prover -max-processes 1
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -max-processes 1
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -max-processes 1
  ```

Tips

You must have at least 4 GB of RAM per processor for analysis. For instance, if your machine has 16 GB of RAM, do not use this option to specify more than four processes.

See Also

Topics
“Specify Polyspace Analysis Options”

-non-preemptable-tasks

Specify functions that represent nonpreemptable tasks

Syntax

`-non-preemptable-tasks` *`function1`*`[,`*`function2`*`[,...]]`

Description

This option affects a Bug Finder analysis only.

`-non-preemptable-tasks` *`function1`*`[,`*`function2`*`[,...]]` specifies functions that represent nonpreemptable tasks.

The functions cannot be interrupted by other noncyclic tasks and cyclic tasks but can be interrupted by interrupts, preemptable or nonpreemptable. Noncyclic tasks are specified with the option `Tasks (-entry-points)`, cyclic tasks with the option `Cyclic tasks (-cyclic-tasks)` and interrupts with the option `Interrupts (-interrupts)`. For examples, see “Define Preemptable Interrupts and Nonpreemptable Tasks”.

To specify a function as a nonpreemptable cyclic task, you must first specify the function as a cyclic or noncyclic task. The functions that you specify must have the prototype:

```
void function_name(void);
```

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

See Also

`-non-preemptable-tasks` | `-preemptable-interrupts` | `Critical section details (-critical-section-begin -critical-section-end)` | `Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics
“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”
“Configuring Polyspace Multitasking Analysis Manually”
“Protections for Shared Variables in Multitasking Code”
“Define Preemptable Interrupts and Nonpreemptable Tasks”
“Concurrency Defects”

Introduced in R2016b

-options-file

Run Polyspace using list of options

Syntax

`-options-file` *`file`*

Description

`-options-file` *`file`* specifies a file which lists your analysis options. The file must be a text file with each option on a separate line. Use `#` to add comments to this file.

Examples

1. Create an options file called `listofoptions.txt` with your options. For example:

   - Bug Finder or Bug Finder Server:

     ```
     #These are the options for MyBugFinderProject
     -lang c
     -prog MyBugFinderProject
     -author jsmith
     -sources "mymain.c,funAlgebra.c,funGeometry.c"
     -target x86_64
     -compiler generic
     -dos
     -misra2 required-rules
     -do-not-generate-results-for all-headers
     -checkers default
     -disable-checkers concurrency
     -results-dir C:\Polyspace\MyBugFinderProject
     ```

   - Code Prover or Code Prover Server:

     ```
     #These are the options for MyCodeProverProject
     -lang c
     -prog MyCodeProverProject
     -author jsmith
     -sources "mymain.c,funAlgebra.c,funGeometry.c"
     -target x86_64
     -compiler generic
     -dos
     -misra2 required-rules
     -do-not-generate-results-for all-headers
     -main-generator
     -results-dir C:\Polyspace\MyCodeProverProject
     ```

2. Run Polyspace using options in the file `listofoptions.txt`:

   - Bug Finder:

     ```
     polyspace-bug-finder -options-file listofoptions.txt
     ```

   - Code Prover:

```
polyspace-code-prover -options-file listofoptions.txt
```

- Bug Finder Server:

  ```
  polyspace-bug-finder-server -options-file listofoptions.txt
  ```

- Code Prover Server:

  ```
  polyspace-code-prover-server -options-file listofoptions.txt
  ```

See Also

Topics
“Specify Polyspace Analysis Options”

-options-for-sources

Specify analysis options specific to a source file

Syntax

`-options-for-sources` *`filename options`*

Description

`-options-for-sources` *`filename options`* associates a semicolon-separated list of Polyspace analysis options with the source file specified by *`filename`*..

This option is primarily used when the `polyspace-configure` command creates an options file for the subsequent Polyspace analysis. The option `-options-for-sources` associates a group of analysis options such as include folders and macro definitions with specific source files.

However, you can directly enter this option when manually writing options files. This option is useful in situations where you want to associate a group of options with a specific source file without applying it to other files.

In the user interface of the Polyspace desktop products, you can create a Polyspace project from your build command. The project uses the option `-options-for-sources` to associate specific Polyspace analysis options with specific files. However, when you open the project in the user interface, you cannot see the use of this option. Open the project in a text editor to see this option.

Examples

In this sample options file, the include folder `/usr/lib/gcc/x86_64-linux-gnu/6/include` and the macros `__STDC_VERSION__` and `__GNUC__` are associated only with the source file `file.c` and not `fileAnother.c`.

```
-options-for-sources file.c;-I /usr/lib/gcc/x86_64-linux-gnu/6/include;
-options-for-sources file.c;-D __STDC_VERSION__=201112L;-D __GNUC__=6;
-sources file.c
-sources fileAnother.c
```

For the options used in this example, see:

- `-sources`
- `-I`
- `Preprocessor definitions (-D)`

See Also

`-options-file` | `polyspace-configure`

Topics

“Specify Polyspace Analysis Options”

-preemptable-interrupts

Specify functions that represent preemptable interrupts

Syntax

`-preemptable-interrupts` *`function1`*`[,`*`function2`*`[,...]]`

Description

This option affects a Bug Finder analysis only.

`-preemptable-interrupts` *`function1`*`[,`*`function2`*`[,...]]` specifies functions that represent preemptable interrupts.

The function acts as an interrupt in every way except that it can be interrupted by other interrupts, preemptable or nonpreemptable. Interrupts are specified with the option `Interrupts (-interrupts)`. For examples, see “Define Preemptable Interrupts and Nonpreemptable Tasks”.

To specify a function as a preemptable interrupt, you must first specify the function as an interrupt. The functions that you specify must have the prototype:

```
void function_name(void);
```

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

See Also

`-non-preemptable-tasks` | `-preemptable-interrupts` | `Critical section details (-critical-section-begin -critical-section-end)` | `Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics
“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”
“Configuring Polyspace Multitasking Analysis Manually”
“Protections for Shared Variables in Multitasking Code”
“Define Preemptable Interrupts and Nonpreemptable Tasks”
“Concurrency Defects”

Introduced in R2016b

-prog

Specify name of project

Syntax

`-prog` *`projectName`*

Description

`-prog` *`projectName`* specifies a name for your Polyspace project. This name must use only letters, numbers, underscores (_), dashes (-), or periods (.).

The name appears in the analysis log and a few other places.

Examples

Assign a name to your Polyspace project:

- Bug Finder:

  ```
  polyspace-bug-finder -prog MyApp
  ```
- Code Prover:

  ```
  polyspace-code-prover -prog MyApp
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -prog MyApp
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -prog MyApp
  ```

See Also

`-author` | `-date`

Topics

“Specify Polyspace Analysis Options”

-regex-replace-rgx -regex-replace-fmt

Make replacements in preprocessor directives

Syntax

`-regex-replace-rgx` *`matchFileName`* `-regex-replace-fmt` *`replacementFileName`*

Description

`-regex-replace-rgx` *`matchFileName`* `-regex-replace-fmt` *`replacementFileName`* replaces tokens in preprocessor directives for the purposes of Polyspace analysis. The original source code is unchanged. You match a token using a regular expression in the file *`matchFileName`* and replace the token using a replacement in the file *`replacementFileName`*.

Use this option only to replace or remove tokens in the preprocessor directives *before preprocessing*. If a token in your source code causes a compilation error, you can typically replace or remove the token from the preprocessed code. Use the more convenient option `Command/script to apply to preprocessed files (-post-preprocessing-command)`. You cannot make the replacements in preprocessed code only for tokens in preprocessor directives.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

In the user interface, specify absolute paths to the text files with the search and replace patterns.

Examples

Suppose you want to replace `_rom_beg` in this `#define` directive:

```
#define ROM_BEG_ADDR (uint16*)(&_rom_beg)
```

and modify the directive to:

```
#define ROM_BEG_ADDR (0x4000u)
```

Specify this regular expression in a file `match.txt`:

```
^\s*#define\s+ROM_BEG_ADDR\s+\(uint16_t\*\)\(\&_rom_beg\)
```

These elements are used in the regular expression:

- `^` asserts position at the start of a line.
- `\s*` represents zero or more whitespace characters.
- `\s+` represents one or more whitespace characters.

The characters *, &, (and) in the original expression are escaped with `\`. For a complete list of regular expressions, see Perl documentation.

Specify the replacement in a file `replace.txt`.

```
#define ROM_BEG_ADDR \(0x4000u\)
```

Specify the two text files during analysis with the options `-regex-replace-rgx` and `-regex-replace-fmt`:

- Bug Finder:

```
polyspace-bug-finder -sources filename
                                -regex-replace-rgx match.txt
                                -regex-replace-fmt replace.txt
```

- Code Prover:

```
polyspace-code-prover -sources filename
                                -regex-replace-rgx match.txt
                                -regex-replace-fmt replace.txt
```

- Bug Finder Server:

```
polyspace-bug-finder-server -sources filename
                                -regex-replace-rgx match.txt
                                -regex-replace-fmt replace.txt
```

- Code Prover Server:

```
polyspace-code-prover-server -sources filename
                                -regex-replace-rgx match.txt
                                -regex-replace-fmt replace.txt
```

See Also

`Command/script to apply to preprocessed files (-post-preprocessing-command)`

Topics

“Specify Polyspace Analysis Options”

-report-output-name

Specify name of report

Syntax

`-report-output-name` *`reportName`*

Description

`-report-output-name` *`reportName`* specifies the name of an analysis report.

The default name for a report is *`Prog_Template.Format`*:

- *`Prog`* is the name of the project specified by `-prog`.
- *`TemplateName`* is the type of report template specified by `-report-template`.
- *`Format`* is the file extension for the report specified by `-report-output-format`.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

Specify the name of the analysis report:

- Bug Finder:

```
polyspace-bug-finder -report-template Developer
      -report-output-name Airbag_v3.doc
```

- Code Prover:

```
polyspace-code-prover -report-template Developer
      -report-output-name Airbag_v3.doc
```

- Bug Finder Server:

```
polyspace-bug-finder-server -report-template Developer
      -report-output-name Airbag_v3.doc
```

- Code Prover Server:

```
polyspace-code-prover-server -report-template Developer
      -report-output-name Airbag_v3.doc
```

See Also

`Bug Finder and Code Prover report (-report-template)` | `Output format (-report-output-format)`

Topics

“Specify Polyspace Analysis Options”
“Generate Reports”

-results-dir

Specify the results folder

Syntax

```
-results-dir resultsFolder
```

Description

`-results-dir resultsFolder` specifies where to save the analysis results. The default location at the command line is the current folder.

If you are running analysis in the user interface of the Polyspace desktop products, see “Run Polyspace Analysis on Desktop”.

Examples

Specify to store your results in the `RESULTS` folder:

- Bug Finder:

  ```
  polyspace-bug-finder -results-dir RESULTS
  ```

- Code Prover:

  ```
  polyspace-code-prover -results-dir RESULTS
  ```

- Bug Finder Server:

  ```
  polyspace-bug-finder-server -results-dir RESULTS
  ```

- Code Prover Server:

  ```
  polyspace-code-prover-server -results-dir RESULTS
  ```

You can create the name of the results folder based on the verification date and time. For instance, in a Bash shell, enter these commands to create a variable `RESULTS` that begins with `results_` and contains the current date and time:

```
export DATETIME=$(date +%d%B_%HH%M_%A)
export RESULTS=results_$DATE
```

You can then use the variable `RESULTS` as argument of the option `-results-dir`:

```
-results-dir $RESULTS
```

See Also

Topics
“Specify Polyspace Analysis Options”

-scheduler

Specify cluster or job scheduler

Syntax

`-scheduler` *`schedulingOption`*

Description

`-scheduler` *`schedulingOption`* specifies the head node of the MATLAB Parallel Server cluster that manages Polyspace analysis submissions from multiple clients and allocates the analysis to worker nodes. You use this option along with the option `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)` to offload an analysis from a desktop to a remote cluster. Note that you use this option with the commands in the desktop products (`polyspace-bug-finder` and `polyspace-code-prover`) and not the commands in the server products (`polyspace-bug-finder-server` and `polyspace-code-prover-server`).

For more information, see “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

Examples

Run a batch analysis on a remote server using one of these syntaxes for the job scheduler:

- Bug Finder:

  ```
  polyspace-bug-finder -batch -scheduler NodeHost
  polyspace-bug-finder -batch -scheduler 192.168.1.124:12400
  polyspace-bug-finder -batch -scheduler MJSName@NodeHost
  ```

- Code Prover:

  ```
  polyspace-code-prover -batch -scheduler NodeHost
  polyspace-code-prover -batch -scheduler 192.168.1.124:12400
  polyspace-code-prover -batch -scheduler MJSName@NodeHost
  ```

For details, see “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”.

You can track the status of the job using the `polyspace-jobs-manager` command:

```
polyspace-jobs-manager listjobs -scheduler NodeHost
```

See Also

`Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`

Topics

“Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”
“Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”

-sources

Specify source files

Syntax

```
-sources file1[,file2,...]
-sources file1 -sources file2
```

Description

`-sources file1[,file2,...]` or `-sources file1 -sources file2` specifies the list of source files that you want to analyze. You can use standard UNIX wildcards with this option to specify your sources.

The source files are compiled in the order in which they are specified.

Examples

Analyze the files `mymain.c`, `funAlgebra.c`, and `funGeometry.c`.

- Bug Finder:

  ```
  polyspace-bug-finder -sources mymain.c
       -sources funAlgebra.c -sources funGeometry.c
  ```

- Code Prover:

  ```
  polyspace-code-prover -sources mymain.c
       -sources funAlgebra.c -sources funGeometry.c
  ```

- Bug Finder Server:

  ```
  polyspace-bug-finder-server -sources mymain.c
       -sources funAlgebra.c -sources funGeometry.c
  ```

- Code Prover Server:

  ```
  polyspace-code-prover-server -sources mymain.c
       -sources funAlgebra.c -sources funGeometry.c
  ```

See Also

`-sources-list-file` | `polyspace-configure`

Topics

“Specify Polyspace Analysis Options”

-sources-list-file

Specify file containing list of sources

Syntax

`-sources-list-file` *`file_path`*

Description

`-sources-list-file` *`file_path`* specifies the absolute path to a text file that lists each file name that you want to analyze.

To specify your sources in the text file, on each line, specify the path to a source file. You can specify an absolute path or a path relative to the folder from which you are running the analysis. For example:

```
C:\Sources\myfile.c
C:\Sources2\myfile2.c
```

Examples

Run analysis on files listed in `files.txt`:

- Bug Finder:

  ```
  polyspace-bug-finder -sources-list-file "C:\Analysis\files.txt"
  polyspace-bug-finder -sources-list-file "/home/polyspace/files.txt"
  ```

- Code Prover:

  ```
  polyspace-code-prover -sources-list-file "C:\Analysis\files.txt
  polyspace-code-prover -sources-list-file "/home/polyspace/files.txt"
  ```

- Bug Finder Server:

  ```
  polyspace-bug-finder-server -sources-list-file "C:\Analysis\files.txt"
  polyspace-bug-finder-server -sources-list-file "/home/polyspace/files.txt"
  ```

- Code Prover Server:

  ```
  polyspace-code-prover-server -sources-list-file "C:\Analysis\files.txt
  polyspace-code-prover-server -sources-list-file "/home/polyspace/files.txt"
  ```

See Also

Topics
“Specify Polyspace Analysis Options”

-submit-job-from-previous-compilation-results

Specify that the analysis job must be resubmitted without recompilation

Syntax

```
-submit-job-from-previous-compilation-results
```

Description

`-submit-job-from-previous-compilation-results` specifies that the Polyspace analysis must start after the compilation phase with compilation results from a previous analysis. The option is primarily useful when offloading a Polyspace analysis from desktops to remote servers. If a remote analysis stops after compilation, for instance because of communication problems between the server and client computers, use this option. Note that you use this option with the commands in the desktop products (`polyspace-bug-finder` and `polyspace-code-prover`) and not the commands in the server products (`polyspace-bug-finder-server` and `polyspace-code-prover-server`).

When you perform a remote analysis:

1. On the local host computer, the Polyspace software performs code compilation and coding rule checking.
2. The analysis job is then submitted to the MATLAB job scheduler on the head node of the MATLAB Parallel Server cluster.
3. The head node of the MATLAB Parallel Server cluster assigns the verification job to a worker node, where the remaining phases of the Polyspace analysis occur.

If an analysis stops after completing the first step and you restart the analysis, use this option to reuse compilation results from the previous analysis. You thereby avoid restarting the analysis from the compilation phase.

If previous compilation results do not exist in the current folder, an error occurs. Remove the option and restart analysis from the compilation phase.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

Specify remote analysis with compilation results from a previous analysis:

- Bug Finder:

  ```
  polyspace-bug-finder -batch -scheduler localhost
         -submit-job-from-previous-compilation-results
  ```

- Code Prover:

  ```
  polyspace-code-prover -batch -scheduler localhost
         -submit-job-from-previous-compilation-results
  ```

See Also

Topics
“Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”
“Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”

-termination-functions

Specify process termination functions

Syntax

`-termination-functions` *`function1`*`[,`*`function2`*`[,...]]`

Description

`-termination-functions` *`function1`*`[,`*`function2`*`[,...]]` specifies functions that behave like the exit function and terminate your program.

Use this option to specify program termination functions that are declared but not defined in your code.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

Polyspace detects an **Integer division by zero** defect in the following code because it does not recognize that `my_exit` terminates the program.

```
void my_exit();

double reciprocal(int val) {
  if(val==0)
    my_exit();
  return (1/val);
}
```

To prevent Polyspace from flagging the division operation, use the `-termination-functions` option:

```
polyspace-bug-finder -termination-functions my_exit
```

See Also

`polyspaceBugFinder`

Topics

“Run Polyspace Analysis from Command Line”

-tmp-dir-in-results-dir

Keep temporary files in results folder

Syntax

```
-tmp-dir-in-results-dir
```

Description

`-tmp-dir-in-results-dir` specifies that temporary files must be stored in a subfolder of the results folder. Use this option only when the standard temporary folder does not have enough disk space. If the results folder is mounted on a network drive, this option can slow down your processor.

To learn how Polyspace determines the temporary folder location, see “Storage of Temporary Files”.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

Store temporary files in the results folder:

- Bug Finder:

  ```
  polyspace-bug-finder -tmp-dir-in-results-dir
  ```
- Code Prover:

  ```
  polyspace-code-prover -tmp-dir-in-results-dir
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -tmp-dir-in-results-dir
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -tmp-dir-in-results-dir
  ```

See Also

Topics
“Specify Polyspace Analysis Options”

-v | -version

Display Polyspace version number

Syntax

```
-v
-version
```

Description

`-v` or `-version` displays the version number of your Polyspace product.

Examples

Display the version number and release of your Polyspace product:

- Bug Finder:

  ```
  polyspace-bug-finder -v
  ```
- Code Prover:

  ```
  polyspace-code-prover -v
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -v
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -v
  ```

-xml-annotations-description

Apply custom code annotations to Polyspace analysis results

Syntax

`-xml-annotations-description` *`file_path`*

Description

`-xml-annotations-description` *`file_path`* uses the annotation syntax defined in the XML file located in *`file_path`* to interpret code annotations in your source files. You can use the XML file to specify an annotation syntax and map it to the Polyspace annotation syntax. When you run an analysis by using this option, you can justify and hide results with annotations that use your syntax. If you run Polyspace at the command line, *`file_path`* is the absolute path or path relative to the folder from which you run the command. If you run Polyspace through the user interface, *`file_path`* is the absolute path.

If you are running an analysis through the user interface, you can enter this option in the **Other** field, under the **Advanced Settings** node on the **Configuration** pane. See `Other`.

Why Use This Option

If you have existing annotations from previous code reviews, you can import these annotations to Polyspace. You do not have to review and justify results that you have already annotated. Similarly, if your code comments must adhere to a specific format, you can map and import that format to Polyspace.

Examples

Import Existing Annotations for Coding Rule Violations

Suppose that you have previously reviewed source file `zero_div.c` containing the following code, and justified certain MISRA C: 2012 violations by using custom annotations.

```
#include <stdio.h>

/* Violation of Misra C:2012
rules 8.4 and 8.7 on the next
line of code. */

int func(int p) //My_rule 50, 51
{
    int i;
    int j = 1;

    i = 1024 / (j - p);
    return i;
}

/* Violation of Misra C:2012
rule 8.4 on the next line of
code */

int main(void){ //My_rule 50
    int x=func(2);
    return x;
}
```

The code comments **`My_rule 50, 51`** and **`My_rule 50`** do not use the Polyspace annotation syntax. Instead, you use a convention where you place all MISRA rules in a single numbered list. In this list, rules 8.4 and 8.7 correspond to the numbers 50 and 51.You can check this code for MISRA C: 2012 violations by typing the command:

- Bug Finder:

  ```
  polyspace-bug-finder -sources source_path -misra3 all
  ```

- Code Prover:

  ```
  polyspace-code-prover -sources source_path -misra3 all
  ```

- Bug Finder Server:

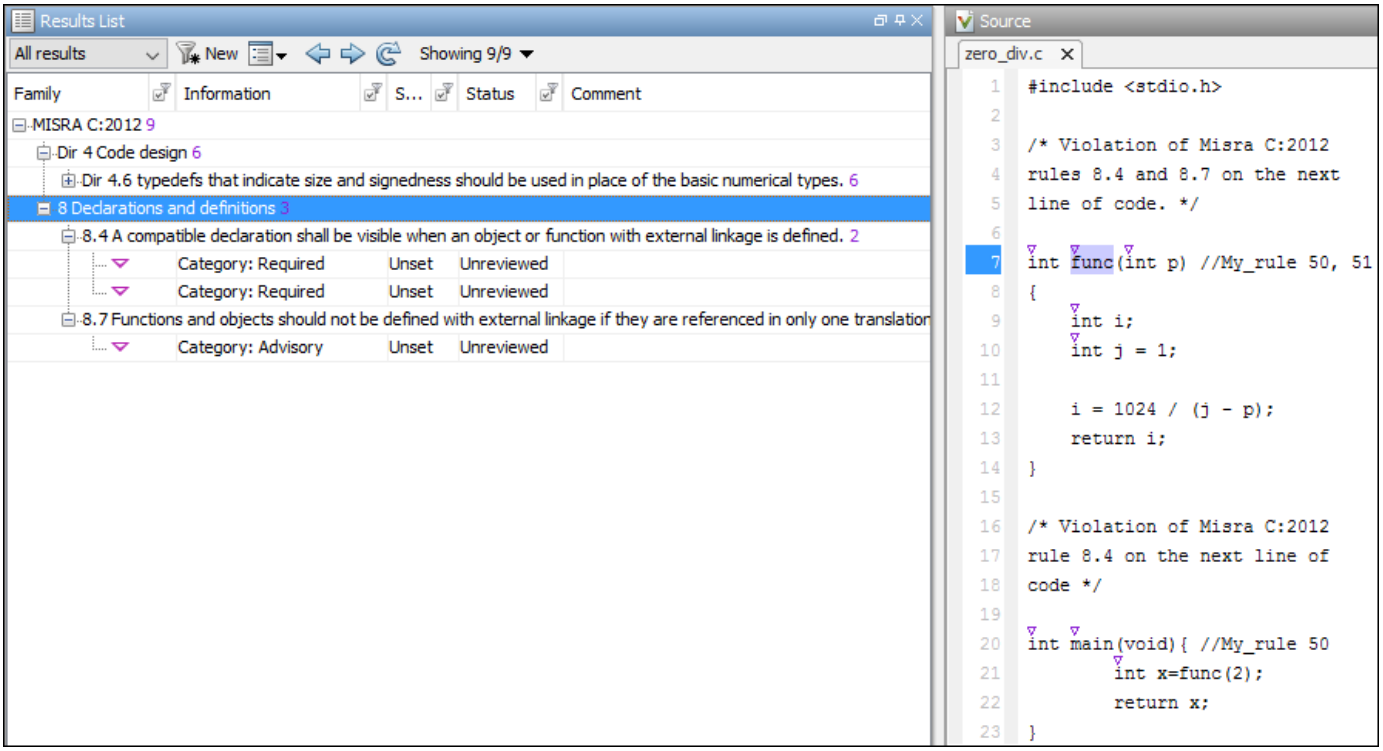
  ```
  polyspace-bug-finder-server -sources source_path -misra3 all
  ```

- Code Prover Server:

  ```
  polyspace-code-prover-server -sources source_path -misra3 all
  ```

`source_path` is the path to `zero_div.c`.

The annotated violations appear in the **Results List** pane. You must review and justify them again.



This XML example defines the annotation format used in `zero_div.c` and maps it to the Polyspace annotation syntax:

- The format of the annotation is the keyword `My_rule`, followed by a space and one or more comma-separated alphanumeric rule identifiers.
- Rule identifiers 50 and 51 are mapped to MISRA C: 2012 rules 8.4 and 8.7 respectively. The mapping uses the Polyspace annotation syntax.

```
<?xml version="1.0" encoding="UTF-8"?>

<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
             Group="example annotation">

  <Expressions Search_For_Keywords="My_rule"
               Separator_Result_Name="," >


    <!-- This section defines the annotation syntax format -->
   <Expression Mode="SAME_LINE"
               Regex="My_rule\s(\w+(\s*,\s*\w+)*)"
               Rule_Identifier_Position="1"
               />

 </Expressions>
  <!-- This section maps the user annotation to the Polyspace
  annotation syntax -->
 <Mapping>
 <Result_Name_Mapping  Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
 <Result_Name_Mapping  Rule_Identifier="51" Family="MISRA-C3" Result_Name="8.7"/>
```

```
 </Mapping>
</Annotations>
```

To import the existing annotations and apply them to the corresponding Polyspace results:

1. Copy the preceding code example to a text editor and save it on your machine as `annotations_description.xml`, for instance in `C:\Polyspace_workspace\annotations\`.
2. Rerun the analysis on `zero_div.c` by using the command:

   - Bug Finder:

     ```
     polyspace-bug-finder -sources source_path -misra3 all ^
     -xml-annotations-desription ^
     C:\Polyspace_workspace\annotations\annotations_description.xml
     ```

   - Code Prover:

     ```
     polyspace-code-prover -sources source_path -misra3 all ^
     -xml-annotations-desription ^
     C:\Polyspace_workspace\annotations\annotations_description.xml
     ```

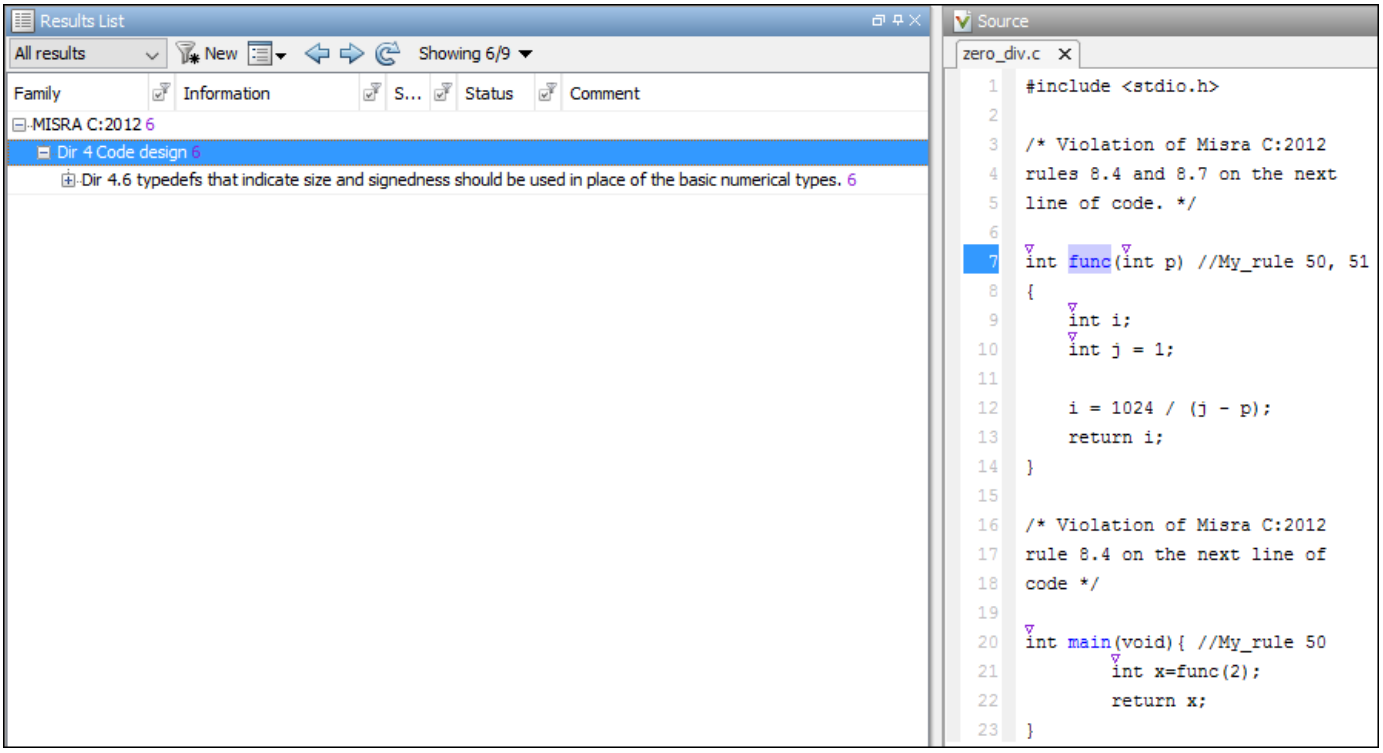
   - Bug Finder Server:

     ```
     polyspace-bug-finder-server -sources source_path -misra3 all ^
     -xml-annotations-desription ^
     C:\Polyspace_workspace\annotations\annotations_description.xml
     ```

   - Code Prover Server:

     ```
     polyspace-code-prover-server -sources source_path -misra3 all ^
     -xml-annotations-desription ^
     C:\Polyspace_workspace\annotations\annotations_description.xml
     ```

Polyspace considers the annotated results justified and hides them in the **Results List** pane.



See Also

Topics

“Specify Polyspace Analysis Options”
“Define Custom Annotation Format”
“Annotation Description Full XML Template”

Introduced in R2017b

Defects

Numerical Defects

Absorption of float operand

One addition or subtraction operand is absorbed by the other operand

Description

This defect occurs when one operand of an addition or subtraction operation is *always* negligibly small compared to the other operand. Therefore, the result of the operation is always equal to the value of the larger operand, making the operation redundant.

Risk

Redundant operations waste execution cycles of your processor.

The absorption of a float operand can indicate design issues elsewhere in the code. It is possible that the developer expected a different range for one of the operands and did not expect the redundancy of the operation. However, the operand range is different from what the developer expects because of issues elsewhere in the code.

Fix

See if the operand ranges are what you expect. To see the ranges, place your cursor on the operation.

- If the ranges are what you expect, justify why you have the redundant operation in place. For instance, the code is only partially written and you anticipate other values for one or both of the operands from future unwritten code.

  If you cannot justify the redundant operation, remove it.
- If the ranges are not what you expect, in your code, trace back to see where the ranges come from. To begin your traceback, search for instances of the operand in your code. Browse through previous instances of the operand and determine where the unexpected range originates.

To determine when one operand is negligible compared to the other operand, the defect uses rules based on IEEE 754 standards. To fix the defect, instead of using the actual rules, you can use this heuristic: the ratio of the larger to the smaller operand must be less than 2^{p-1} at least for some values. Here, `p` is equal to 24 for 32-bit precision and 53 for 64-bit precision. To determine the precision, the defect uses your specification for `Target processor type (-target)`.

This defect appears only if one operand is *always* negligibly smaller than the other operand. To see instances of subnormal operands or results, use the check **Subnormal Float** in Polyspace Code Prover.

Examples

One Addition Operand Negligibly Smaller Than The Other Operand

```
#include <stdlib.h>

float get_signal(void);
void do_operation(float);

float input_signal1(void) {
```

```
    float temp = get_signal();
    if(temp > 0. && temp < 1e-30)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

float input_signal2(void) {
    float temp = get_signal();
    if(temp > 1.)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

void main() {
    float signal1 = input_signal1();
    float signal2 = input_signal2();
    float super_signal = signal1 + signal2;
    do_operation(super_signal);
}
```

In this example, the defect appears on the addition because the operand `signal1` is in the range `(0,1e-30)` but `signal2` is greater than `1`.

Correction — Remove Redundant Operation

One possible correction is to remove the redundant addition operation. In the following corrected code, the operand `signal2` and its associated code is also removed from consideration.

```
#include <stdlib.h>

float get_signal(void);
void do_operation(float);

float input_signal1(void) {
    float temp = get_signal();
    if(temp > 0. && temp < 1e-30)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

void main() {
    float signal1 = input_signal1();
    do_operation(signal1);
}
```

Correction — Verify Operand Range

Another possible correction is to see if the operand ranges are what you expect. For instance, if one of the operand range is not supposed to be negligibly small, fix the issue causing the small range. In

the following corrected code, the range `(0,1e-2)` is imposed on `signal2` so that it is not *always* negligibly small as compared to `signal1`.

```
#include <stdlib.h>

float get_signal(void);
void do_operation(float);

float input_signal1(void) {
    float temp = get_signal();
    if(temp > 0. && temp < 1e-2)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

float input_signal2(void) {
    float temp = get_signal();
    if(temp > 1.)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

void main() {
    float signal1 = input_signal1();
    float signal2 = input_signal2();
    float super_signal = signal1 + signal2;
    do_operation(super_signal);
}
```

Result Information

Group: Numerical
Language: C | C++
Default: On
Command-Line Syntax: `FLOAT_ABSORPTION`
Impact: High
CWE ID: 189, 682, 873

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2016b

Bitwise operation on negative value

Undefined behavior for bitwise operations on negative values

Description

This defect occurs when bitwise operators (`>>`, `^`, `|`, `~`, `but`, `not`, `&`) are used on signed integer variables with negative values.

Risk

If the value of the signed integer is negative, bitwise operation results can be unexpected because:

- Bitwise operations on negative values can produce compiler-specific results.
- Unexpected calculations can lead to additional vulnerabilities, such as buffer overflow.

Fix

When performing bitwise operations, use `unsigned` integers to avoid unexpected results.

Examples

Right-Shift of Negative Integer

```
#include <stdio.h>
#include <stdarg.h>

static void demo_sprintf(const char *format, ...)
{
    int rc;
    va_list ap;
    char buf[sizeof("256")];

    va_start(ap, format);
    rc = vsprintf(buf, format, ap);
    if (rc == -1 || rc >= sizeof(buf)) {
        /* Handle error */
    }
    va_end(ap);
}

void bug_bitwiseneg()
{
    int stringify = 0x80000000;
    demo_sprintf("%u", stringify >> 24);
}
```

In this example, the statement `demo_sprintf("%u", stringify >> 24)` stops the program unexpectedly. You expect the result of `stringify >> 24` to be `0x80`. However, the actual result is `0xffffff80` because `stringify` is signed and negative. The sign bit is also shifted.

Correction — Add `unsigned` Keyword

By adding the `unsigned` keyword, `stringify` is not negative and the right-shift operation gives the expected result of `0x80`.

```
#include <stdio.h>
#include <stdarg.h>

static void demo_sprintf(const char *format, ...)
{
    int rc;
    va_list ap;
    char buf[sizeof("256")];

    va_start(ap, format);
    rc = vsprintf(buf, format, ap);
    if (rc == -1 || rc >= sizeof(buf)) {
        /* Handle error */
    }
    va_end(ap);
}

void corrected_bitwiseneg()
{
    unsigned int stringify = 0x80000000;
    demo_sprintf("%u", stringify >> 24);
}
```

Result Information

Group: Numerical
Language: C | C++
Default: Off
Command-Line Syntax: `BITWISE_NEG`
Impact: Medium
CWE ID: 682, 758

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2016b

Float conversion overflow

Overflow when converting between floating point data types

Description

This defect occurs when converting a floating point number to a smaller floating point data type. If the variable does not have enough memory to represent the original number, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Overflows can result in unpredictable values from computations. The result can be infinity or the maximum finite value depending on the rounding mode used in the implementation. If you use the result of an overflowing conversion in subsequent computations and do not account for the overflow, you can see unexpected results.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variable being converted acquires its current value You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller floating point types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Converting from `double` to `float`

```
float convert(void) {

    double diam = 1e100;
    return (float)diam;
}
```

In the return statement, the variable `diam` of type double (64 bits) is converted to a variable of type float (32 bits). However, the value 1^100 requires more than 32 bits to be precisely represented.

Check Information

Group: Numerical
Language: C | C++
Default: On
Command-Line Syntax: `FLOAT_CONV_OVFL`
Impact: High
CWE ID: 189, 197, 681

See Also

`Find defects (-checkers)` | `Integer conversion overflow` | `Sign change integer conversion overflow` | `Unsigned integer conversion overflow`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2013b

Float division by zero

Dividing floating point number by zero

Description

This defect occurs when the denominator of a division operation can be a zero-valued floating point number.

Risk

A division by zero can result in a program crash.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the denominator variable acquires a zero value. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

It is a good practice to check for zero values of a denominator before division and handle the error. Instead of performing the division directly:

```
res = num/den;
```

use a library function that handles zero values of the denominator before performing the division:

```
res = div(num, den);
```

See examples of fixes below.

If you do not want to fix the issue, for instance, when you handle infinities in your code, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Dividing a Floating Point Number by Zero

```
float fraction(float num)
{
    float denom = 0.0;
    float result = 0.0;

    result = num/denom;

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

Correction — Check Before Division

```
float fraction(float num)
{
    float denom = 0.0;
    float result = 0.0;

    if( ((int)denom) != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
float fraction(float num)
{
    float denom = 2.0;
    float result = 0.0;

    result = num/denom;

    return result;
}
```

Check Information

Group: Numerical
Language: C | C++
Default: On
Command-Line Syntax: `FLOAT_ZERO_DIV`
Impact: High
CWE ID: 189, 369

See Also

`Find defects (-checkers)` | `Integer division by zero`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2013b

Float overflow

Overflow from operation between floating points

Description

This defect occurs when an operation on floating point variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Overflows can result in unpredictable values from computations. The result can be infinity or the maximum finite value depending on the rounding mode used in the implementation. If you use the result of an overflowing computation in subsequent computations and do not account for the overflow, you can see unexpected results.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, for instance, when you handle infinities in your code, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Multiplication of Floats

```
#include <float.h>

float square(void) {

   float val = FLT_MAX;
   return val * val;
}
```

In the return statement, the variable `val` is multiplied by itself. The square of the maximum float value cannot be represented by a float (the return type for this function) because the value of `val` is the maximum float value.

Correction — Different Storage Type

One possible correction is to store the result of the operation in a larger data type. In this example, by returning a `double` instead of a `float`, the overflow defect is fixed.

```
#include <float.h>

double square(void) {
    float val = FLT_MAX;

    return (double)val * (double)val;
}
```

Check Information

Group: Numerical
Language: C | C++
Default: Off
Command-Line Syntax: `FLOAT_OVFL`
Impact: Low
CWE ID: 189, 682, 873

See Also

`Find defects (-checkers)` | `Integer overflow` | `Unsigned integer overflow`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2013b

Integer constant overflow

Constant value falls outside range of integer data type

Description

This defect occurs when you assign a compile-time constant to a signed integer variable whose data type cannot accommodate the value. An n-bit signed integer holds values in the range $[-2^{n-1}, 2^{n-1}-1]$.

For instance, `c` is an 8-bit signed `char` variable that cannot hold the value 255.

```
signed char c = 255;
```

To determine the sizes of fundamental types, Bug Finder uses your specification for `Target processor type (-target)`.

Risk

The default behavior for constant overflows can vary between compilers and platforms. Retaining constant overflows can reduce the portability of your code.

Even if your compilers wraps around overflowing constants with a warning, the wrap-around behavior can be unintended and cause unexpected results.

Fix

Check if the constant value is what you intended. If the value is correct, use a different, possibly wider, data type for the variable.

Examples

Overflowing Constant from Macro Expansion

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
    char c1 = MAX_UNSIGNED_CHAR;
    char c2 = MAX_SIGNED_CHAR+1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow. To reproduce these defects, use a `Target processor type (-target)` where `char` is signed by default.

Correction — Use Different Data Type

One possible correction is to use a different data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
```

```
    unsigned char c1 = MAX_UNSIGNED_CHAR;
    unsigned char c2 = MAX_SIGNED_CHAR+1;
}
```

Result Information

Group: Numerical
Language: C | C++
Default: Off
Command-Line Syntax: `INT_CONSTANT_OVFL`
Impact: Medium
CWE ID: 128, 189, 190, 191

See Also

`Find defects (-checkers)` | `Integer conversion overflow` | `Integer overflow` | `Sign change integer conversion overflow` | `Unsigned integer constant overflow` | `Unsigned integer conversion overflow` | `Unsigned integer overflow`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Integer conversion overflow

Overflow when converting between integer types

Description

This defect occurs when converting an integer to a smaller integer type. If the variable does not have enough bytes to represent the original value, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Integer conversion overflows result in undefined behavior.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Converting from `int` to `char`

```
char convert(void) {

    int num = 1000000;

    return (char)num;
}
```

In the return statement, the integer variable `num` is converted to a `char`. However, an 8-bit or 16-bit character cannot represent 1000000 because it requires at least 20 bits. So the conversion operation overflows.

Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number.

```
long convert(void) {

    int num = 1000000;

    return (long)num;
}
```

Check Information

Group: Numerical
Language: C | C++
Default: On
Command-Line Syntax: `INT_CONV_OVFL`
Impact: High
CWE ID: 128, 189, 190, 191, 192, 197

See Also

`Float conversion overflow` | `Unsigned integer conversion overflow` | `Sign change integer conversion overflow` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2013b

Integer division by zero

Dividing integer number by zero

Description

This defect occurs when the denominator of a division or modulo operation can be a zero-valued integer.

Risk

A division by zero can result in a program crash.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the denominator variable acquires a zero value. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

It is a good practice to check for zero values of a denominator before division and handle the error. Instead of performing the division directly:

```
res = num/den;
```

use a library function that handles zero values of the denominator before performing the division:

```
res = div(num, den);
```

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Dividing an Integer by Zero

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    result = num/denom;

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

Correction — Check Before Division

```
int fraction(int num)
{
```

```
    int denom = 0;
    int result = 0;

    if (denom != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
int fraction(int num)
{
    int denom = 2;
    int result = 0;

    result = num/denom;

    return result;
}
```

Modulo Operation with Zero

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        arr[i] = input % i;
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

In this example, Polyspace flags the modulo operation as a division by zero. Because modulo is inherently a division operation, the divisor (right hand argument) cannot be zero. The modulo operation uses the `for` loop index as the divisor. However, the `for` loop starts at zero, which cannot be an iterator.

Correction — Check Divisor Before Operation

One possible correction is checking the divisor before the modulo operation. In this example, see if the index `i` is zero before the modulo operation.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        if(i != 0)
        {
             arr[i] = input % i;
        }
```

```
        else
        {
             arr[i] = input;
        }
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

Correction — Change Divisor

Another possible correction is changing the divisor to a nonzero integer. In this example, add one to the index before the `%` operation to avoid dividing by zero.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
         arr[i] = input % (i+1);
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

Check Information

Group: Numerical
Language: C | C++
Default: On
Command-Line Syntax: `INT_ZERO_DIV`
Impact: High
CWE ID: 189, 369

See Also

`Find defects (-checkers)` | `Float division by zero`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2013b

Integer overflow

Overflow from operation between integers

Description

This defect occurs when an operation on integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Integer overflows on signed integers result in undefined behavior.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

To avoid overflows in general, try one of these techniques:

- Keep integer variable values restricted to within half the range of signed integers.
- In operations that might overflow, check for conditions that can lead to the overflow and implement wrap around or saturation behavior depending on how the result of the operation is used. The result then becomes predictable and can be safely used in subsequent computations.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Addition of Maximum Integer

```
#include <limits.h>

int plusplus(void) {

    int var = INT_MAX;
    var++;
```

```
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

Correction — Different Storage Type

One possible correction is to change data types. Store the result of the operation in a larger data type (Note that on a 32-bit machine, `int` and `long` has the same size). In this example, on a 32-bit machine, by returning a `long long` instead of an `int`, the overflow error is fixed.

```
#include <limits.h>

long long plusplus(void) {

    long long lvar = INT_MAX;
    lvar++;
    return lvar;
}
```

Check Information

Group: Numerical
Language: C | C++
Default: Off
Command-Line Syntax: `INT_OVFL`
Impact: Medium
CWE ID: 128, 189, 190, 191, 192

See Also

`Find defects (-checkers)` | `Float overflow` | `Unsigned integer overflow`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2013b

Integer precision exceeded

Operation using integer size instead of precision can cause undefined behavior

Description

This defect occurs when an integer expression uses the integer size in an operation that exceeds the integer precision. On some architectures, the size of an integer in memory can include sign and padding bits. On these architectures, the integer size is larger than the precision which is just the number of bits that represent the value of the integer.

Risk

Using the size of an integer in an operation on the integer precision can result in integer overflow, wrap around, or unexpected results. For instance, an unsigned integer can be stored in memory in 64 bits, but uses only 48 bits to represent its value. A 56 bits left-shift operation on this integer is undefined behavior.

Assuming that the size of an integer is equal to its precision can also result in program portability issues between different architectures.

Fix

Do not use the size of an integer instead of its precision. To determine the integer precision, implement a precision computation routine or use a builtin function such as `__builtin_popcount()`.

Examples

Using Size of `unsigned int` for Left Shift Operation

```
#include <limits.h>

unsigned int func(unsigned int exp)
{
    if (exp >= sizeof(unsigned int) * CHAR_BIT) {
        /* Handle error */
    }
    return 1U << exp;
}
```

In this example, the function uses a left shift operation to return the value of 2 raised to the power of `exp`. The operation shifts the bits of `1U` by `exp` positions to the left. The `if` statement ensures that the operation does not shift the bits by a number of positions `exp` greater than the size of an `unsigned int`. However, if `unsigned int` contains padding bits, the value returned by `sizeof()` is larger than the precision of `unsigned int`. As a result, some values of `exp` might be too large, and the shift operation might be undefined behavior.

Correction — Implement Function to Compute Precision of `unsigned int`

One possible correction is to implement a function `popcount()` that computes the precision of `unsigned int` by counting the number of set bits.

```
#include <stddef.h>
#include <stdint.h>
#include <limits.h>

size_t popcount(uintmax_t);
#define PRECISION(umax_value) popcount(umax_value)


unsigned int func(unsigned int exp)
{
    if (exp >= PRECISION(UINT_MAX)) {
        /* Handle error */
    }
    return 1 << exp;
}




size_t popcount(uintmax_t num)
{
    size_t precision = 0;
    while (num != 0) {
        if (num % 2 == 1) {
            precision++;
        }
        num >>= 1;
    }
    return precision;
}
```

Result Information

Group: Numerical
Language: C | C++
Default: Off
Command-Line Syntax: INT_PRECISION_EXCEEDED
Impact: Low
CWE ID: 190

See Also

Bitwise operation on negative value | Find defects (-checkers) | Integer conversion overflow | Integer overflow | MISRA C:2012 Rule 10.1 | MISRA C:2012 Rule 10.2 | Possible invalid operation on boolean operand | Shift of a negative value | Shift operation overflow | Unsigned integer conversion overflow | Unsigned integer overflow

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Invalid use of standard library floating point routine

Wrong arguments to standard library function

Description

This defect occurs when you use invalid arguments with a floating point function from the standard library. This defect picks up:

- Rounding and absolute value routines

  ```
  ceil, fabs, floor, fmod
  ```

- Fractions and division routines

  ```
  fmod, modf
  ```

- Exponents and log routines

  ```
  frexp, ldexp, sqrt, pow, exp, log, log10
  ```

- Trigonometry function routines

  ```
  cos, sin, tan, acos, asin, atan, atan2, cosh, sinh, tanh, acosh, asinh,
  atanh
  ```

Risk

Domain errors on standard library floating point functions result in implementation-defined values. If you use the function return value in subsequent computations, you can see unexpected results.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the function argument acquires invalid values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

It is a good practice to handle for domain errors before using a standard library floating point function. For instance, before calling the `acos` function, check if the argument is in [-1.0, 1.0] and handle the error.

See examples of fixes below.

If you do not want to fix the issue, for instance, when you handle infinities in your code, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Arc Cosine Operation

```
#include <math.h>

double arccosine(void) {
    double degree = 5.0;
    return acos(degree);
}
```

The input value to `acos` must be in the interval `[-1,1]`. This input argument, `degree`, is outside this range.

Correction — Change Input Argument

One possible correction is to change the input value to fit the specified range. In this example, change the input value from degrees to radians to fix this defect.

```
#include <math.h>

double arccosine(void) {
    double degree = 5.0;
    double radian = degree * 3.14159 / 180.;
    return acos(radian);
}
```

Check Information

Group: Numerical
Language: C | C++
Default: On
Command-Line Syntax: `FLOAT_STD_LIB`
Impact: High
CWE ID: 227, 369, 682, 873

See Also

`Find defects (-checkers)` | `Invalid use of standard library integer routine` | `Invalid use of standard library memory routine` | `Invalid use of standard library routine` | `Invalid use of standard library string routine`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2013b

Invalid use of standard library integer routine

Wrong arguments to standard library function

Description

This defect occurs when you use invalid arguments with an integer function from the standard library. This defect picks up:

- Character Conversion

  `toupper, tolower`
- Character Checks

  `isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit`
- Integer Division

  `div, ldiv`
- Absolute Values

  `abs, labs`

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Absolute Value of Large Negative

```
#include <limits.h>
#include <stdlib.h>

int absoluteValue(void) {

    int neg = INT_MIN;
    return abs(neg);
}
```

The input value to `abs` is `INT_MIN`. The absolute value of `INT_MIN` is `INT_MAX+1`. This number cannot be represented by the type `int`.

Correction — Change Input Argument

One possible correction is to change the input value to fit returned data type. In this example, change the input value to `INT_MIN+1`.

```
#include <limits.h>
#include <stdlib.h>

int absoluteValue(void) {

    int neg = INT_MIN+1;
    return abs(neg);
}
```

Check Information

Group: Numerical
Language: C | C++
Default: On
Command-Line Syntax: `INT_STD_LIB`
Impact: High
CWE ID: 227, 369, 682, 872

See Also

`Find defects (-checkers)` | `Invalid use of standard library floating point routine` | `Invalid use of standard library memory routine` | `Invalid use of standard library routine` | `Invalid use of standard library string routine`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2013b

Possible invalid operation on boolean operand

Operation can exceed precision of Boolean operand or result in arbitrary value

Description

This defect occurs when you use a Boolean operand in an arithmetic, relational, or bitwise operation and:

- The Boolean operand has a trap representation. The size of a Boolean type in memory is at least one addressable unit (size of `char`). A Boolean type requires only one bit to represent the value `true (1)` or `false (0)`. The representation of a Boolean operand in memory contains padding bits. The memory representation can result in values that are not `true` or `false`, a trap representation.
- The result of the operation can exceed the precision of the Boolean operand.

For example, in this code snippet:

```
bool_v >> 2
```

- If the value of `bool_v` is `true (1)` or `false (0)`, the bitwise shift exceeds the one-bit precision of `bool_v` and always results in `0`.
- If `bool_v` has a trap representation, the result of the operation is an arbitrary value.

Possible invalid operation on boolean operand raises no defect when:

- The operation does not result in a precision overflow. For instance, bitwise & or `|` operations with `0x01` or `0x00`.
- The Boolean operand cannot have a trap representation. For instance, a constant expression that results in `0` or `1`, or a comparison evaluated to `true` or `false`.

Risk

Arithmetic, relational, or bitwise operations on a Boolean operand can exceed the operand precision and cause unexpected results when used as a Boolean value. Operations on Boolean operands with trap representations can return arbitrary values.

Fix

Avoid performing operations on Boolean operands other than these operations:

- Assignment operation (=).
- Equality operations (== or `!=`).
- Logical operations (&&, `||`, or `!`).

Examples

Possible Trap Representation of Boolean Operand

```
#include <stdio.h>
#include <stdbool.h>
```

```
#define BOOL _Bool

int arr[2] = {1, 2};

int func(BOOL b)
{
    return arr[b];
}

int main(void)
{
    BOOL b;
    char* ptr = (char*)&b;
    *ptr = 64;
    return func(b);
}
```

In this example, Boolean operand `b` is used as an array index in `func` for an array with two elements. Depending on the compiler and optimization flags you use, the value `b` might not be `0` or `1`. For instance, in Linux Debian 8, if you use `gcc` version 4.9 with optimization flag `-O0`, the value of `b` is 64, which causes a buffer overflow.

Correction — Use Only Last Significant Bit Value of Boolean Operand

One possible correction is to use a variable `b0` of type `unsigned int` to get only the value of the last significant bit of the Boolean operand. The value of this bit is in the range `[0..1]`, even if the Boolean operand has a trap representation.

```
#include <stdio.h>
#include <stdbool.h>

#define BOOL _Bool

int arr[2] = {1, 2};

int func(BOOL b)
{
    unsigned int b0 = (unsigned int)b;
    b0 &= 0x1;
    return arr[b0];
}

int main(void)
{
    BOOL b;
    char* ptr = (char*)&b;
    *ptr = 64;
    return func(b);
}
```

Note that a trap representation is often the result of an earlier issue in the code, such as:

- A non-initialized variable of bool type.
- A side effect that modifies any part of a bool type object using a lvalue expression.
- A read of a bool member from a union type with the last stored value of another type.

As such, it is best practice to respect boolean semantics even in C++ code.

<= Operation Uses Boolean Operands

```
#include <iostream>

template <typename T>
bool less_or_equal(const T& x, const T& y)
{
    std::cout << "INTEGER VERSION" << '\n';
    return x <= y;
}
bool b1 = true, b2 = false;
int i1 = 2, i2 = 3;

int main()
{
    std::cout << std::boolalpha;
    std::cout << "less_or_equal(" << b1 << ',' << b2 << ") = " << less_or_equal<bool>(b1, b2) << '\n';
    std::cout << "less_or_equal(" << i1 << ',' << i2 << ") = " << less_or_equal<int>(11, 12) << '\n';
    return 0;
}
```

In this example, function template `less_or_equal` evaluates whether variable x is less than or equal to y. When you pass boolean types to this function, the `<=` operation might result in an arbitrary value if the memory representation of the operands, including their padding bits, is neither 1 nor 0.

Correction — Specialize Function Template for Boolean Types

One possible correction is to specialize the function template for boolean types. The specialized function template uses a logical (`||`) operation to compare the boolean operands.

```
#include <iostream>

template <typename T>
bool less_or_equal(const T& x, const T& y)
{
    std::cout << "INTEGER VERSION" << '\n';
    return x <= y;
}

template<>
bool less_or_equal<bool>(const bool& x, const bool& y)
{
    std::cout << "BOOLEAN VERSION" << '\n';
    return !x || y;
}

bool b1 = true, b2 = false;
int i1 = 2, i2 = 3;

int main()
{
    std::cout << std::boolalpha;
    std::cout << "less_or_equal(" << b1 << ',' << b2 << ") = " << less_or_equal<bool>(b1, b2) << '\n';
    std::cout << "less_or_equal(" << i1 << ',' << i2 << ") = " << less_or_equal<int>(11, 12) << '\n';
    return 0;
}
```

Result Information

Group: Numerical
Language: C | C++
Default: Off
Command-Line Syntax: `INVALID_OPERATION_ON_BOOLEAN`
Impact: Low
CWE ID: 190

See Also

`Bitwise and arithmetic operation on the same data` | `Bitwise operation on negative value` | `Find defects (-checkers)` | `Integer overflow` | `Integer conversion overflow` | `Integer precision exceeded` | `MISRA C++:2008 Rule 4-5-2` | `MISRA C:2004 Rule 12.6` | `MISRA C:2012 Rule 10.1` | `MISRA C:2012 Rule 12.2` | `Shift of a negative value` | `Shift operation overflow` | `Unsigned integer overflow` | `Unsigned integer conversion overflow`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Precision loss in integer to float conversion

Least significant bits of integer lost during conversion to floating-point type

Description

This defect occurs when you cast an integer value to a floating-point type that cannot represent the original integer value.

For instance, the `long int` value `1234567890L` is too large for a variable of type `float` .

Risk

If the floating-point type cannot represent the integer value, the behavior is undefined (see C11 standard, 6.3.1.4, paragraph 2). For instance, least significant bits of the variable value can be dropped leading to unexpected results.

Fix

Convert to a floating-point type that can represent the integer value.

For instance, if the `float` data type cannot represent the integer value, use the `double` data type instead.

When writing a function that converts an integer to floating point type, before the conversion, check if the integer value can be represented in the floating-point type. For instance, `DBL_MANT_DIG * log2(FLT_RADIX)` represents the number of base-2 digits in the type `double`. Before conversion to the type `double`, check if this number is greater than or equal to the precision of the integer that you are converting. To determine the precision of an integer `num`, use this code:

```
size_t precision = 0;
while (num != 0) {
   if (num % 2 == 1) {
     precision++;
   }
   num >>= 1;
}
```

Some implementations provide a builtin function to determine the precision of an integer. For instance, GCC provides the function `__builtin_popcount`.

Examples

Conversion of Large Integer to Floating-Point Type

```
#include <stdio.h>

int main(void) {
  long int big = 1234567890L;
  float approx = big;
  printf("%ld\n", (big - (long int)approx));
  return 0;
}
```

In this example, the `long int` variable `big` is converted to `float`.

Correction — Use a Wider Floating-Point Type

One possible correction is to convert to the `double` data type instead of `float`.

```
#include <stdio.h>

int main(void) {
  long int big = 1234567890L;
  double approx = big;
  printf("%ld\n", (big - (long int)approx));
  return 0;
}
```

Result Information

Group: Numerical
Language: C | C++
Default: Off
Command-Line Syntax: `INT_TO_FLOAT_PRECISION_LOSS`
Impact: Low
CWE ID: 189, 681, 704

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Shift of a negative value

Shift operator on negative value

Description

This defect occurs when a bit-wise shift is used on a variable that can have negative values.

Risk

Shifts on negative values overwrite the sign bit that identifies a number as negative. The shift operation can result in unexpected values.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variable being shifted acquires negative values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

To fix the defect, check for negative values before the bit-wise shift operation and perform appropriate error handling.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Shifting a negative variable

```
int shifting(int val)
{
    int res = -1;
    return res << val;
}
```

In the return statement, the variable `res` is shifted a certain number of bits to the left. However, because `res` is negative, the shift might overwrite the sign bit.

Correction — Change the Data Type

One possible correction is to change the data type of the shifted variable to unsigned. This correction eliminates the sign bit, so left shifting does not change the sign of the variable.

```
int shifting(int val)
{
    unsigned int res = -1;
    return res << val;
}
```

Check Information

Group: Numerical
Language: C | C++
Default: Off
Command-Line Syntax: `SHIFT_NEG`
Impact: Low
CWE ID: 189

See Also

`Find defects (-checkers)` | `Shift operation overflow`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2013b

Shift operation overflow

Overflow from shifting operation

Description

This defect occurs when a shift operation can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different data types depends on your processor. See `Target processor type (-target)`.

Risk

Shift operation overflows can result in undefined behavior.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the shift operation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the shift operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Left Shift of Integer

```
int left_shift(void) {

    int foo = 33;
    return 1 << foo;
}
```

In the return statement of this function, bit-wise shift operation is performed shifting 1 `foo` bits to the left. However, an `int` has only 32 bits, so the range of the shift must be between 0 and 31. Therefore, this shift operation causes an overflow.

Correction — Different storage type

One possible correction is to store the shift operation result in a larger data type. In this example, by returning a `long long` instead of an `int`, the overflow defect is fixed.

```
long long left_shift(void) {

    int foo = 33;
    return 1LL << foo;
}
```

Check Information

Group: Numerical
Language: C | C++
Default: Off
Command-Line Syntax: `SHIFT_OVFL`
Impact: Low
CWE ID: 189, 190

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2013b

Sign change integer conversion overflow

Overflow when converting between signed and unsigned integers

Description

This defect occurs when converting an unsigned integer to a signed integer. If the variable does not have enough bytes to represent both the original constant and the sign bit, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Convert from `unsigned char` to `char`

```
char sign_change(void) {
    unsigned char count = 255;

    return (char)count;
}
```

In the return statement, the unsigned character variable `count` is converted to a signed character. However, `char` has 8 bits, 1 for the sign of the constant and 7 to represent the number. The conversion operation overflows because 255 uses 8 bits.

Correction — Change conversion types

One possible correction is using a larger integer type. By using an `int`, there are enough bits to represent the sign and the number value.

```
int sign_change(void) {
    unsigned char count = 255;

    return (int)count;
}
```

Check Information

Group: Numerical
Language: C | C++

Default: On
Command-Line Syntax: `SIGN_CHANGE`
Impact: Medium
CWE ID: 192, 194, 195, 196

See Also

`Find defects (-checkers)|Float conversion overflow|Integer conversion overflow|Unsigned integer conversion overflow`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2013b

Unsigned integer constant overflow

Constant value falls outside range of unsigned integer data type

Description

This defect occurs when you assign a compile-time constant to a unsigned integer variable whose data type cannot accommodate the value. An `n`-bit unsigned integer holds values in the range `[0, 2^n-1]`.

For instance, `c` is an 8-bit unsigned `char` variable that cannot hold the value 256.

```
unsigned char c = 256;
```

To determine the sizes of fundamental types, Bug Finder uses your specification for `Target processor type (-target)`.

Risk

The C standard states that overflowing unsigned integers must be wrapped around (see, for instance, the C11 standard, section 6.2.5). However, the wrap-around behavior can be unintended and cause unexpected results.

Fix

Check if the constant value is what you intended. If the value is correct, use a wider data type for the variable.

Examples

Overflowing Constant from Macro Expansion

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_UNSIGNED_SHORT 65535

void main() {
    unsigned char c1 = MAX_UNSIGNED_CHAR + 1;
    unsigned short c2 = MAX_UNSIGNED_SHORT + 1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow.

Correction — Use Wider Data Type

One possible correction is to use a wider data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_UNSIGNED_SHORT 65535

void main() {
    unsigned short c1 = MAX_UNSIGNED_CHAR + 1;
    unsigned int c2 = MAX_UNSIGNED_SHORT + 1;
}
```

Result Information

Group: Numerical
Language: C | C++
Default: Off
Command-Line Syntax: `UINT_CONSTANT_OVFL`
Impact: Low
CWE ID: 128, 189, 190, 191

See Also

`Find defects (-checkers)` | `Integer constant overflow` | `Integer conversion overflow` | `Integer overflow` | `Sign change integer conversion overflow` | `Unsigned integer conversion overflow` | `Unsigned integer overflow`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Unsigned integer conversion overflow

Overflow when converting between unsigned integer types

Description

This defect occurs when converting an unsigned integer to a smaller unsigned integer type. If the variable does not have enough bytes to represent the original constant, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Integer conversion overflows result in undefined behavior.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Converting from `int` to `char`

```
unsigned char convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned char)unum;
}
```

In the return statement, the unsigned integer variable `unum` is converted to an unsigned character type. However, the conversion overflows because 1000000 requires at least 20 bits. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `unum` is reduced by modulo `2^8` because a character data type can only represent `2^8-1`.

Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number. For example, `long`.

```
unsigned long convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned long)unum;
}
```

Check Information

Group: Numerical
Language: C | C++
Default: On
Command-Line Syntax: `UINT_CONV_OVFL`
Impact: Low
CWE ID: 128, 131, 189, 190, 191, 192, 197

See Also

`Find defects (-checkers)` | `Float conversion overflow` | `Integer conversion overflow` | `Sign change integer conversion overflow`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2013b

Unsigned integer overflow

Overflow from operation between unsigned integers

Description

This defect occurs when an operation on unsigned integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

The C11 standard states that unsigned integer overflows result in wrap-around behavior. However, a wrap around behavior might not always be desirable. For instance, if the result of a computation is used as an array size and the computation overflows, the array size is much smaller than expected.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling. In the error handling code, you can override the default wrap-around behavior for overflows and implement saturation behavior, for instance.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Add One to Maximum Unsigned Integer

```
#include <limits.h>

unsigned int plusplus(void) {

    unsigned uvar = UINT_MAX;
    uvar++;
    return uvar;
}
```

In the third statement of this function, the variable `uvar` is increased by 1. However, the value of `uvar` is the maximum unsigned integer value, so 1 plus the maximum integer value cannot be

represented by an `unsigned int`. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `uvar` is reduced by modulo `UINT_MAX`. The result is `uvar = 1`.

Correction — Different Storage Type

One possible correction is to store the operation result in a larger data type. In this example, by returning an `unsigned long long` instead of an `unsigned int`, the overflow error is fixed.

```
#include <limits.h>

unsigned long long plusplus(void) {

    unsigned long long ullvar = UINT_MAX;
    ullvar++;
    return ullvar;
}
```

Check Information

Group: Numerical
Language: C | C++
Default: Off
Command-Line Syntax: `UINT_OVFL`
Impact: Low
CWE ID: 128, 131, 189, 190, 191, 192

See Also

`Find defects (-checkers)` | `Float overflow` | `Integer overflow`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2013b

Use of plain char type for numerical value

Plain `char` variable in arithmetic operation without explicit signedness

Description

This defect occurs when `char` variables without explicit signedness are used in these ways:

- To store non-`char` constants.
- In an arithmetic operation when the `char` is:
  - A negative value.
  - The result of a sign changing overflow.
- As a buffer offset.

`char` variables without a `signed` or `unsigned` qualifier can be signed or unsigned depending on your compiler.

Risk

Operations on a plain char can result in unexpected numerical values. If the char is used as an offset, the char can cause buffer overflow or underflow.

Fix

When initializing a char variable, to avoid implementation-defined confusion, explicitly state whether the char is signed or unsigned.

Examples

Divide by `char` Variable

```
#include <stdio.h>

void badplaincharuse(void)
{
    char c = 200;
    int i = 1000;
    (void)printf("i/c = %d\n", i/c);
}
```

In this example, the char variable `c` can be signed or unsigned depending on your compiler. Assuming 8-bit, two's complement character types, the result is either `i/c = 5` (unsigned char) or `i/c = -17` (signed char). The correct result is unknown without knowing the signedness of `char`.

Correction — Add `signed` Qualifier

One possible correction is to add a `signed` qualifier to `char`. This clarification makes the operation defined.

```
#include <stdio.h>

void badplaincharuse(void)
```

```
{
    signed char c = -56;
    int i = 1000;
    (void)printf("i/c = %d\n", i/c);
}
```

Result Information

Group: Numerical
Language: C | C++
Default: Off
Command-Line Syntax: BAD_PLAIN_CHAR_USE
Impact: Medium
CWE ID: 682, 758

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2016b

Static Memory Defects

Arithmetic operation with NULL pointer

Arithmetic operation performed on NULL pointer

Description

This defect occurs when an arithmetic operation involves a pointer whose value is NULL.

Risk

Performing pointer arithmetic on a null pointer and dereferencing the resulting pointer is undefined behavior. In most implementations, the dereference can cause your program to crash.

Fix

Check a pointer for NULL before arithmetic operations on the pointer.

If the issue occurs despite an earlier check for NULL, look for intermediate events between the check and the subsequent dereference. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

Examples

Arithmetic Operation with NULL Pointer Error

```
#include<stdlib.h>

int Check_Next_Value(int *loc, int val)
 {
  int *ptr = loc, found = 0;

  if (ptr==NULL)
   {
      ptr++;
      /* Defect: NULL pointer shifted */

      if (*ptr==val) found=1;
   }

  return(found);
 }
```

When ptr is a NULL pointer, the code enters the if statement body. Therefore, a NULL pointer is shifted in the statement ptr++.

Correction — Avoid NULL Pointer Arithmetic

One possible correction is to perform the arithmetic operation when ptr is not NULL.

```
#include<stdlib.h>
```

```
int Check_Next_Value(int *loc, int val)
 {
  int *ptr = loc, found = 0;

  /* Fix: Perform operation when ptr is not NULL */
  if (ptr!=NULL)
   {
      ptr++;

      if (*ptr==val) found=1;
   }

  return(found);
 }
```

Check Information

Group: Static memory
Language: C | C++
Default: Off
Command-Line Syntax: `NULL_PTR_ARITH`
Impact: Low

See Also

`Find defects (-checkers)` | `Null pointer`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Array access out of bounds

Array index outside bounds during array access

Description

This defect occurs when an array index falls outside the range `[0...array_size-1]` during array access.

Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Array Access Out of Bounds Error

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
```

```
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

Correction — Keep Array Index Within Array Bounds

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

Check Information

Group: Static memory
Language: C | C++
Default: On
Command-Line Syntax: `OUT_BOUND_ARRAY`
Impact: High
CWE ID: 119, 131, 466

See Also

`Find defects (-checkers)` | `Pointer access out of bounds`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2013b

Buffer overflow from incorrect string format specifier

String format specifier causes buffer argument of standard library functions to overflow

Description

This defect occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

Risk

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

Fix

Use a format specifier that is compatible with the memory buffer size.

Examples

Memory Buffer Overflow

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf);
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

Correction — Use Smaller Precision in Format Specifier

One possible correction is to use a smaller precision in the format specifier.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

Result Information

Group: Static memory
Language: C | C++
Default: On
Command-Line Syntax: `STR_FORMAT_BUFFER_OVERFLOW`
Impact: High
CWE ID: 124, 125, 126, 127

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Destination buffer overflow in string manipulation

Function writes to buffer at offset greater than buffer size

Description

This defect occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

Examples

Buffer Overflow in `sprintf` Use

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>
```

```
void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

Result Information

Group: Static memory
Language: C | C++
Default: On
Command-Line Syntax: STRLIB_BUFFER_OVERFLOW
Impact: High
CWE ID: 121, 125, 135, 251, 787

See Also

`Destination buffer underflow in string manipulation` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Destination buffer underflow in string manipulation

Function writes to buffer at a negative offset from beginning of buffer

Description

This defect occurs when certain string manipulation functions write to their destination buffer argument at a negative offset from the beginning of the buffer.

For instance, for the function `sprintf(char* buffer, const char* format)`, you obtain the `buffer` from an operation `buffer = (char*)arr; ... buffer += offset;`. `arr` is an array and `offset` is a negative value.

Risk

Buffer underflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer underflow also introduces the risk of code injection.

Fix

If the destination buffer argument results from pointer arithmetic, see if you are decrementing a pointer. Fix the pointer decrement by modifying either the original value before decrement or the decrement value.

Examples

Buffer Underflow in `sprintf` Use

```
#include <stdio.h>
#define offset -2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

In this example, `&buffer[offset]` is at a negative offset from the memory allocated to `buffer`.

Correction — Change Pointer Decrementer

One possible correction is to change the value of `offset`.

```
#include <stdio.h>
#define offset 2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";
```

```
    sprintf(&buffer[offset], fmt_string);
}
```

Result Information

Group: Static memory
Language: C | C++
Default: On
Command-Line Syntax: `STRLIB_BUFFER_UNDERFLOW`
Impact: High
CWE ID: 124, 786, 787

See Also

`Destination buffer overflow in string manipulation` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Invalid use of standard library memory routine

Standard library memory function called with invalid arguments

Description

This defect occurs when a memory library function is called with invalid arguments. For instance, the `memcpy` function copies to an array that cannot accommodate the number of bytes copied.

Risk

Use of a memory library function with invalid arguments can result in issues such as buffer overflow.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Invalid Use of Standard Library Memory Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string `str2` is 5, but six characters of string `str1` are copied into `str2` using the `memcpy` function.

Correction — Call Function with Valid Arguments

One possible correction is to adjust the size of `str2` so that it accommodates the characters copied with the `memcpy` function.

```
#include <string.h>
#include <stdio.h>
```

```
char* Copy_First_Six_Letters(void)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  return str2;
 }
```

Check Information

Group: Static memory
Language: C | C++
Default: On
Command-Line Syntax: MEM_STD_LIB
Impact: High
CWE ID: 120, 227, 690

See Also

Find defects (-checkers) | Invalid use of standard library string routine

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Invalid use of standard library string routine

Standard library string function called with invalid arguments

Description

This defect occurs when a string library function is called with invalid arguments.

Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Invalid Use of Standard Library String Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_String(void)
{
 char *res;
 char gbuffer[5],text[20]="ABCDEFGHIJKL";

 res=strcpy(gbuffer,text);
 /* Error: Size of text is less than gbuffer */

 return(res);
}
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

Check Information

Group: Static memory
Language: C | C++
Default: On
Command-Line Syntax: `STR_STD_LIB`
Impact: High
CWE ID: 120, 227, 690

See Also

`Find defects (-checkers)` | `Invalid use of standard library memory routine`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Move operation on const object

`std::move` function is called with object declared `const` or `const&`

Description

This defect occurs when the `std::move` function is called with an object declared `const` or `const&`.

Risk

For objects declared `const` or `const&`, unlike what you might expect, the copy constructor is called instead of the move constructor.

Fix

Avoid calling the `std::move` function on `const` objects. If you want to perform a move operation, cast the `const` object to a non-`const` one and then move the non-`const` object.

Check Information

Group: Programming
Language: C++
Default: On
Command-Line Syntax: `MOVE_CONST_OBJECT`
Impact: High

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Null pointer

NULL pointer dereferenced

Description

This defect occurs when you use a pointer with a value of NULL as if it points to a valid memory location.

Risk

Dereferencing a null pointer is undefined behavior. In most implementations, the dereference can cause your program to crash.

Fix

Check a pointer for NULL before dereference.

If the issue occurs despite an earlier check for NULL, look for intermediate events between the check and the subsequent dereference. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

Examples

Null pointer error

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 int* p=NULL;

 *p=arr[0];
 /* Defect: Null pointer dereference */

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

The pointer p is initialized with value of NULL. However, when the value arr[0] is written to *p, p is assumed to point to a valid memory location.

Correction — Assign Address to Null Pointer Before Dereference

One possible correction is to initialize p with a valid memory address before dereference.

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 /* Fix: Assign address to null pointer */
 int* p=&arr[0];

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

Check Information

Group: Static memory
Language: C | C++
Default: On
Command-Line Syntax: NULL_PTR
Impact: High
CWE ID: 476, 690

See Also

Arithmetic operation with NULL pointer | Find defects (-checkers) | Non-initialized pointer

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

Introduced in R2013b

Pointer access out of bounds

Pointer dereferenced outside its bounds

Description

This defect occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }
```

```
 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

Check Information

Group: Static memory
Language: C | C++
Default: On
Command-Line Syntax: `OUT_BOUND_PTR`
Impact: High
CWE ID: 119, 131, 188, 466, 823

See Also

`Array access out of bounds` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Pointer or reference to stack variable leaving scope

Pointer to local variable leaves the variable scope

Description

This defect occurs when a pointer or reference to a local variable leaves the scope of the variable. For instance:

- A function returns a pointer to a local variable.
- A function performs the assignment `globPtr = &locVar`. `globPtr` is a global pointer variable and `locVar` is a local variable.
- A function performs the assignment `*paramPtr = &locVar`. `paramPtr` is a function parameter that is, for instance, an `int**` pointer and `locVar` is a local `int` variable.
- A C++ method performs the assignment `memPtr = &locVar`. `memPtr` is a pointer data member of the class the method belongs to. `locVar` is a variable local to the method.
- (C++11 and later) A function returns a lambda expression object that captures local variables of the function by reference.

The defect also applies to memory allocated using the `alloca` function. The defect does not apply to static, local variables.

Risk

Local variables are allocated an address on the stack. Once the scope of a local variable ends, this address is available for reuse. Using this address to access the local variable value outside the variable scope can cause unexpected behavior.

If a pointer to a local variable leaves the scope of the variable, Polyspace Bug Finder highlights the defect. The defect appears even if you do not use the address stored in the pointer. For maintainable code, it is a good practice to not allow the pointer to leave the variable scope. Even if you do not use the address in the pointer now, someone else using your function can use the address, causing undefined behavior.

Fix

Do not allow a pointer or reference to a local variable to leave the variable scope.

Examples

Pointer to Local Variable Returned from Function

```
void func2(int *ptr) {
    *ptr = 0;
}

int* func1(void) {
    int ret = 0;
```

```
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

In this example, `func1` returns a pointer to local variable `ret`.

In `main`, `ptr` points to the address of the local variable. When `ptr` is accessed in `func2`, the access is illegal because the scope of `ret` is limited to `func1`,

Pointer to Local Variable Escapes Through Lambda Expression

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [&] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

In this example, the `createAdder` function defines a lambda expression `adder` that captures the local variable `addThis` by reference. The scope of `addThis` is limited to the `createAdder` function. When the object returned by `createAdder` is called, a reference to the variable `addThis` is accessed outside its scope. When accessed in this way, the value of `addThis` is undefined.

Correction – Capture Local Variables by Copy in Lambda Expression Instead of Reference

If a function returns a lambda expression object, avoid capturing local variables by reference in the lambda object. Capture the variables by copy instead.

Variables captured by copy have the same lifetime as the lambda object, but variables captured by reference often have a smaller lifetime than the lambda object itself. When the lambda object is used, these variables accessed outside scope have undefined values.

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [=] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

Result Information

Group: Static memory
Language: C | C++
Default: On
Command-Line Syntax: `LOCAL_ADDR_ESCAPE`
Impact: High
CWE ID: 562, 825

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Subtraction or comparison between pointers to different arrays

Subtraction or comparison between pointers causes undefined behavior

Description

This defect occurs when you subtract or compare pointers that are null or that point to elements in different arrays. The relational operators for the comparison are >, <, >=, and <=.

Risk

When you subtract two pointers to elements in the same array, the result is the difference between the subscripts of the two array elements. Similarly, when you compare two pointers to array elements, the result is the positions of the pointers relative to each other. If the pointers are null or point to different arrays, a subtraction or comparison operation is undefined. If you use the subtraction result as a buffer index, it can cause a buffer overflow.

Fix

Before you subtract or use relational operators to compare pointers to array elements, check that they are non-null and that they point to the same array.

Examples

Subtraction Between Pointers to Elements in Different Arrays

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE20 20

size_t func(void)
{
    int nums[SIZE20];
    int end;
    int *next_num_ptr = nums;
    size_t free_elements;
    /* Increment next_num_ptr as array fills */

    /* Subtraction operation is undefined unless array nums
    is adjacent to variable end in memory. */
    free_elements = &end - next_num_ptr;
    return free_elements;
}
```

In this example, the array `nums` is incrementally filled. Pointer subtraction is then used to determine how many free elements remain. Unless `end` points to a memory location one past the last element of `nums`, the subtraction operation is undefined.

Correction — Subtract Pointers to the Same Array

Subtract the pointer to the last element that was filled from the pointer to the last element in the array.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE20 20

size_t func(void)
{
    int nums[SIZE20];
    int *next_num_ptr = nums;
    size_t free_elements;
    /* Increment next_num_ptr as array fills */

    /* Subtraction operation involves pointers to the same array. */
    free_elements = &(nums[SIZE20 - 1]) - next_num_ptr;

    return free_elements + 1;
}
```

Result Information

Group: Static memory
Language: C | C++
Default: On
Command-Line Syntax: `PTR_TO_DIFF_ARRAY`
Impact: High
CWE ID: 469

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Unreliable cast of function pointer

Function pointer cast to another function pointer with different argument or return type

Description

This defect occurs when a function pointer is cast to another function pointer that has different argument or return type.

This defect applies only if the code language for the project is C.

Risk

If you cast a function pointer to another function pointer with different argument or return type and then use the latter function pointer to call a function, the behavior is undefined.

Fix

Avoid a cast between two function pointers with mismatch in argument or return types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Unreliable cast of function pointer error

```
#include <stdio.h>
#include <math.h>
#define PI 3.142

double Calculate_Sum(int (*fptr)(double))
{
    double  sum = 0.0;
    double  y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;

    fp = sin;
    sum = Calculate_Sum(fp);
    /* Defect: fp implicitly cast to int(*) (double) */
```

```
    printf("sum(sin): %f\n", sum);
    return 0;
}
```

The function pointer `fp` is declared as `double (*)(double)`. However in passing it to function `Calculate_Sum`, `fp` is implicitly cast to `int (*)(double)`.

Correction — Avoid Function Pointer Cast

One possible correction is to check that the function pointer in the definition of `Calculate_Sum` has the same argument and return type as `fp`. This step makes sure that `fp` is not implicitly cast to a different argument or return type.

```
#include <stdio.h>
#include <math.h>
# define PI 3.142

/*Fix: fptr has same argument and return type everywhere*/
double Calculate_Sum(double (*fptr)(double))
{
    double  sum = 0.0;
    double y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;


    fp = sin;
    sum = Calculate_Sum(fp);
    printf("sum(sin): %f\n", sum);

    return 0;
}
```

Check Information

Group: Static memory
Language: C | C++
Default: On
Command-Line Syntax: `FUNC_CAST`
Impact: Medium

See Also

`Find defects (-checkers)` | `Unreliable cast of pointer`

Topics
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Unreliable cast of pointer

Pointer implicitly cast to different data type

Description

This defect occurs when a pointer is implicitly cast to a data type different from its declaration type. Such an implicit casting can take place, for instance, when a pointer to data type `char` is assigned the address of an integer.

This defect applies only if the code language for the project is C.

Risk

Casting a pointer to data type different from its declaration type can result in issues such as buffer overflow. If the cast is implicit, it can indicate a coding error.

Fix

Avoid *implicit* cast of a pointer to a data type different from its declaration type.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Unreliable cast of pointer error

```
#include <string.h>

void Copy_Integer_To_String()
{
 int src[]={1,2,3,4,5,6,7,8,9,10};
 char buffer[]="Buffer_Text";
 strcpy(buffer,src);
 /* Defect: Implicit cast of (int*) to (char*) */
}
```

`src` is declared as an `int*` pointer. The `strcpy` statement, while copying to `buffer`, implicitly casts `src` to `char*`.

Correction — Avoid Pointer Cast

One possible correction is to declare the pointer `src` with the same data type as `buffer`.

```
#include <string.h>
 void Copy_Integer_To_String()
{
 /* Fix: Declare src with same type as buffer */
 char *src[10]={"1","2","3","4","5","6","7","8","9","10"};
 char *buffer[10];
```

```
for(int i=0;i<10;i++)
  buffer[i]="Buffer_Text";

for(int i=0;i<10;i++)
  buffer[i]= src[i];
}
```

Check Information

Group: Static memory
Language: C
Default: On
Command-Line Syntax: PTR_CAST
Impact: Medium
CWE ID: 135, 704, 843

See Also

`Find defects (-checkers)|Unreliable cast of function pointer`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Use of automatic variable as putenv-family function argument

putenv-family function argument not accessible outside its scope

Description

This defect occurs when the argument of a `putenv`-family function is a local variable with automatic duration.

Risk

The function `putenv(char *string)` inserts a pointer to its supplied argument into the environment array, instead of making a copy of the argument. If the argument is an automatic variable, its memory can be overwritten after the function containing the `putenv()` call returns. A subsequent call to `getenv()` from another function returns the address of an out-of-scope variable that cannot be dereferenced legally. This out-of-scope variable can cause environment variables to take on unexpected values, cause the program to stop responding, or allow arbitrary code execution vulnerabilities.

Fix

Use `setenv()`/`unsetenv()` to set and unset environment variables. Alternatively, use `putenv`-family function arguments with dynamically allocated memory, or, if your application has no reentrancy requirements, arguments with static duration. For example, a single thread execution with no recursion or interrupts does not require reentrancy. It cannot be called (reentered) during its execution.

Examples

Automatic Variable as Argument of `putenv()`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024

void func(int var)
{
    char env[SIZE1024];
    int retval = sprintf(env, "TEST=%s", var ? "1" : "0");
    if (retval <= 0) {
        /* Handle error */
    }
    /* Environment variable TEST is set using putenv().
    The argument passed to putenv is an automatic variable. */
    retval = putenv(env);
    if (retval != 0) {
        /* Handle error */
    }
```

```
}
```

In this example, `sprintf()` stores the character string `TEST=var` in `env`. The value of the environment variable `TEST` is then set to `var` by using `putenv()`. Because `env` is an automatic variable, the value of `TEST` can change once `func()` returns.

Correction — Use `static` Variable for Argument of `putenv()`

Declare `env` as a static-duration variable. The memory location of `env` is not overwritten for the duration of the program, even after `func()` returns.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024
void func(int var)
{
    /* static duration variable */
    static char env[SIZE1024];
    int retval = sprintf(env,"TEST=%s", var ? "1" : "0");
    if (retval <= 0) {
        /* Handle error */
    }

    /* Environment variable TEST is set using putenv() */
    retval=putenv(env);
    if (retval != 0) {
        /* Handle error */
    }
}
```

Correction — Use `setenv()` to Set Environment Variable Value

To set the value of `TEST` to `var`, use `setenv()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024

void func(int var)
{
    /* Environment variable TEST is set using setenv() */
    int retval = setenv("TEST", var ? "1" : "0", 1);

    if (retval != 0) {
        /* Handle error */
    }
}
```

Result Information

Group: Static memory
Language: C | C++
Default: On

Command-Line Syntax: PUTENV_AUTO_VAR
Impact: High
CWE ID: 562, 686, 825

See Also

Find defects (-checkers) | Pointer or reference to stack variable leaving scope

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Use of path manipulation function without maximum sized buffer checking

Destination buffer of `getwd` or `realpath` is smaller than `PATH_MAX` bytes

Description

This defect occurs when the destination argument of a path manipulation function such as `realpath` or `getwd` has a buffer size less than `PATH_MAX` bytes.

Risk

A buffer smaller than `PATH_MAX` bytes can overflow but you cannot test the function return value to determine if an overflow occurred. If an overflow occurs, following the function call, the content of the buffer is undefined.

For instance, `char *getwd(char *buf)` copies an absolute path name of the current folder to its argument. If the length of the absolute path name is greater than `PATH_MAX` bytes, `getwd` returns `NULL` and the content of `*buf` is undefined. You can test the return value of `getwd` for `NULL` to see if the function call succeeded.

However, if the allowed buffer for `buf` is less than `PATH_MAX` bytes, a failure can occur for a smaller absolute path name. In this case, `getwd` does not return `NULL` even though a failure occurred. Therefore, the allowed buffer for `buf` must be `PATH_MAX` bytes long.

Fix

Possible fixes are:

- Use a buffer size of `PATH_MAX` bytes. If you obtain the buffer from an unknown source, before using the buffer as argument of `getwd` or `realpath` function, make sure that the size is less than `PATH_MAX` bytes.
- Use a path manipulation function that allows you to specify a buffer size.

  For instance, if you are using `getwd` to get the absolute path name of the current folder, use `char *getcwd(char *buf, size_t size);` instead. The additional argument `size` allows you to specify a size greater than or equal to `PATH_MAX`.
- Allow the function to allocate additional memory dynamically, if possible.

  For instance, `char *realpath(const char *path, char *resolved_path);` dynamically allocates memory if `resolved_path` is `NULL`. However, you have to deallocate this memory later using the `free` function.

Examples

Possible Buffer Overflow in Use of `getwd` Function

```
#include <unistd.h>
#include <linux/limits.h>
#include <stdio.h>
```

```
void func(void) {
    char buf[PATH_MAX];
    if (getwd(buf+1)!= NULL)         {
        printf("cwd is %s\n", buf);
    }
}
```

In this example, although the array `buf` has `PATH_MAX` bytes, the argument of `getwd` is `buf + 1`, whose allowed buffer is less than `PATH_MAX` bytes.

Correction — Use Array of Size `PATH_MAX` Bytes

One possible correction is to use an array argument with size equal to `PATH_MAX` bytes.

```
#include <unistd.h>
#include <linux/limits.h>
#include <stdio.h>

void func(void) {
    char buf[PATH_MAX];
    if (getwd(buf)!= NULL)         {
        printf("cwd is %s\n", buf);
    }
}
```

Result Information

Group: Static memory
Language: C | C++
Default: Off
Command-Line Syntax: `PATH_BUFFER_OVERFLOW`
Impact: High
CWE ID: 785

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Wrong allocated object size for cast

Allocated memory does not match destination pointer

Description

This defect occurs during pointer conversion when the pointer's address is misaligned. If a pointer is converted to a different pointer type, the size of the allocated memory must be a multiple of the size of the destination pointer.

Risk

Dereferencing a misaligned pointer has undefined behavior and can cause your program to crash.

Fix

Suppose you convert a pointer `ptr1` to `ptr2`. If `ptr1` points to a buffer of `N` bytes and `ptr2` is a *`type`* * pointer where `sizeof(`*`type`*`)` is `n` bytes, make sure that `N` is an integer multiple of `n`.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Examples

Dynamic Allocation of Pointers

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(13);
    long *dest;

    dest = (long*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to a `long*`. The dynamically allocated memory of `ptr`, 13 bytes, is not a multiple of the size of `dest`, 4 bytes. This misalignment causes the **Wrong allocated object size for cast** defect.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the allocated memory to 12 instead of 13.

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(12);
    long *dest;

    dest = (long*)ptr;
}
```

Static Allocation of Pointers

```
void static_non_align(void){
    char arr[13], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to an `int*` in line 6. `ptr` has a memory size of 13 bytes because the array `arr` has a size of 13 bytes. The size of `dest` is 4 bytes, which is not a multiple of 13. This misalignment causes the **Wrong allocated object size for cast** defect.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the size of the array `arr` to a multiple of 4.

```
void static_non_align(void){
    char arr[12], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr;
}
```

Allocation with a Function

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(13);  //defect
    dest2 = (char*)my_alloc(13); //not a defect
}
```

In this example, the software raises a defect on the conversion of the pointer returned by `my_alloc(13)` to an `int*` in line 11. `my_alloc(13)` returns a pointer with a dynamically allocated size of 13 bytes. The size of `dest1` is 4 bytes, which is not a divisor of 13. This misalignment causes the **Wrong allocated object size for cast** defect. In line 12, the same function call, `my_alloc(13)`, does not call a defect for the conversion to `dest2` because the size of `char*`, 1 byte, a divisor of 13.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the argument for `my_alloc` to a multiple of 4.

```
#include <stdlib.h>
```

```
void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(12);
    dest2 = (char*)my_alloc(13);
}
```

Check Information

Group: Static Memory
Language: C | C++
Default: Off
Command-Line Syntax: `OBJECT_SIZE_MISMATCH`
Impact: High
CWE ID: 704

See Also

`Find defects (-checkers)` | `Unreliable cast of pointer`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Dynamic Memory Defects

Alignment changed after memory reallocation

Memory reallocation changes the originally stricter alignment of an object

Description

This defect occurs when you use `realloc()` to modify the size of objects with strict memory alignment requirements.

Risk

The pointer returned by `realloc()` can be suitably assigned to objects with less strict alignment requirements. A misaligned memory allocation can lead to buffer underflow or overflow, an illegally dereferenced pointer, or access to arbitrary memory locations. In processors that support misaligned memory, the allocation impacts the performance of the system.

Fix

To reallocate memory:

1. Resize the memory block.
   - In Windows, use `_aligned_realloc()` with the alignment argument used in `_aligned_malloc()` to allocate the original memory block.
   - In UNIX/Linux, use the same function with the same alignment argument used to allocate the original memory block.
2. Copy the original content to the new memory block.
3. Free the original memory block.

Note This fix has implementation-defined behavior. The implementation might not support the requested memory alignment and can have additional constraints for the size of the new memory.

Examples

Memory Reallocated Without Preserving the Original Alignment

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE1024 1024

void func(void)
{
    size_t resize = SIZE1024;
    size_t alignment = 1 << 12; /* 4096 bytes alignment */
    int *ptr = NULL;
    int *ptr1;

    /* Allocate memory with 4096 bytes alignment */
```

```
    if (posix_memalign((void **)&ptr, alignment, sizeof(int)) != 0)
    {
        /* Handle error */
      }

    /*Reallocate memory without using the original alignment.
    ptr1 may not be 4096 bytes aligned. */

    ptr1 = (int *)realloc(ptr, sizeof(int) * resize);

    if (ptr1 == NULL)
    {
        /* Handle error */
    }

    /* Processing using ptr1 */

    /* Free before exit */
    free(ptr1);
}
```

In this example, the allocated memory is 4096-bytes aligned. `realloc()` then resizes the allocated memory. The new pointer `ptr1` might not be 4096-bytes aligned.

Correction — Specify the Alignment for the Reallocated Memory

When you reallocate the memory, use `posix_memalign()` and pass the alignment argument that you used to allocate the original memory.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE1024 1024

void func(void)
{
    size_t resize = SIZE1024;
    size_t alignment = 1 << 12; /* 4096 bytes alignment */
    int *ptr = NULL;

    /* Allocate memory with 4096 bytes alignment */
    if (posix_memalign((void **)&ptr, alignment, sizeof(int)) != 0)
    {
        /* Handle error */
    }

    /* Reallocate memory using the original alignment. */
    if (posix_memalign((void **)&ptr, alignment, sizeof(int) * resize) != 0)
    {
        /* Handle error */
        free(ptr);
        ptr = NULL;
    }

    /* Processing using ptr */
```

```
    /* Free before exit */
    free(ptr);
}
```

Result Information

Group: Dynamic memory
Language: C | C++
Default: On
Command-Line Syntax: `ALIGNMENT_CHANGE`
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Deallocation of previously deallocated pointer

Memory freed more than once without allocation

Description

This defect occurs when a block of memory is freed more than once using the `free` function without an intermediate allocation.

Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to free this block of memory can result in a segmentation fault.

Fix

The fix depends on the root cause of the defect. See if you intended to allocate a memory block to the pointer between the first deallocation and the second. Otherwise, remove the second `free` statement.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before freeing pointers, check them for NULL values and handle the error. In this way, you are protected against freeing an already freed block.

Examples

Deallocation of Previously Deallocated Pointer Error

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    free (pi);
    /* Defect: pi has already been freed */
}
```

The first `free` statement releases the block of memory that `pi` refers to. The second `free` statement on `pi` releases a block of memory that has been freed already.

Correction — Remove Duplicate Deallocation

One possible correction is to remove the second `free` statement.

```
#include <stdlib.h>

void allocate_and_free(void)
{
```

```
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    /* Fix: remove second deallocation */
 }
```

Check Information

Group: Dynamic memory
Language: C | C++
Default: On
Command-Line Syntax: DOUBLE_DEALLOCATION
Impact: High
CWE ID: 415, 825

See Also

`Find defects (-checkers)|Use of previously freed pointer`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Invalid deletion of pointer

Pointer deallocation using `delete` without corresponding allocation using `new`

Description

This defect occurs when:

- You release a block of memory with the `delete` operator but the memory was previously not allocated with the `new` operator.
- You release a block of memory with the `delete` operator using the single-object notation but the memory was previously allocated as an array with the `new` operator.

This defect applies only to C++ source files.

Risk

The risk depends on the cause of the issue:

- The `delete` operator releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.
- If you use the single-object notation for `delete` on a pointer that is previously allocated with the array notation for `new`, the behavior is undefined.

The issue can also highlight other coding errors. For instance, you perhaps wanted to use the `delete` operator or a previous `new` operator on a different pointer.

Fix

The fix depends on the cause of the issue:

- In most cases, you can fix the issue by removing the `delete` statement. If the pointer is not allocated memory from the heap with the `new` operator, you do not need to release the pointer with `delete`. You can simply reuse the pointer as required or let the object be destroyed at the end of its scope.
- In case of mismatched notation for `new` and `delete`, correct the mismatch. For instance, to allocate and deallocate a single object, use this notation:

  ```
  classType* ptr = new classType;
  delete ptr;
  ```

  To allocate and deallocate an array objects, use this notation:

  ```
  classType* p2 = new classType[10];
  delete[] p2;
  ```

If the issue highlights a coding error such as use of `delete` or `new` on the wrong pointer, correct the error.

Examples

Deleting Static Memory

```
void assign_ones(void)
{
    int ptr[10];

    for(int i=0;i<10;i++)
        *(ptr+i)=1;

    delete[] ptr;
}
```

The pointer `ptr` is released using the `delete` operator. However, `ptr` points to a memory location that was not dynamically allocated.

Correction: Remove Pointer Deallocation

If the number of elements of the array `ptr` is known at compile time, one possible correction is to remove the deallocation of the pointer `ptr`.

```
void assign_ones(void)
{
    int ptr[10];

    for(int i=0;i<10;i++)
        *(ptr+i)=1;
}
```

Correction — Add Pointer Allocation

If the number of array elements is not known at compile time, one possible correction is to dynamically allocate memory to the array `ptr` using the `new` operator.

```
void assign_ones(int num)
{
    int *ptr = new int[num];

    for(int i=0; i < num; i++)
        *(ptr+i) = 1;

    delete[] ptr;
   }
```

Mismatched `new` and `delete`

```
int main (void)
{
    int *p_scale = new int[5];

    //more code using scal

    delete p_scale;
}
```

In this example, `p_scale` is initialized to an array of size 5 using `new int[5]`. However, `p_scale` is deleted with `delete` instead of `delete[]`. The `new`-`delete` pair does not match. Do not use `delete` without the brackets when deleting arrays.

Correction — Match `delete` to `new`

One possible correction is to add brackets so the `delete` matches the `new []` declaration.

```
int main (void)
{
    int *p_scale = new int[5];

    //more code using p_scale

    delete[] p_scale;
}
```

Correction — Match `new` to `delete`

Another possible correction is to change the declaration of `p_scale`. If you meant to initialize `p_scale` as 5 itself instead of an array of size 5, you must use different syntax. For this correction, change the square brackets in the initialization to parentheses. Leave the `delete` statement as it is.

```
int main (void)
{
    int *p_scale = new int(5);

    //more code using p_scale

    delete p_scale;
}
```

Check Information

Group: Dynamic memory
Language: C++
Default: Off
Command-Line Syntax: `BAD_DELETE`
Impact: High
CWE ID: 404

See Also

`Find defects (-checkers)` | `Invalid free of pointer` | `Memory leak`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Invalid free of pointer

Pointer deallocation without a corresponding dynamic allocation

Description

This defect occurs when a block of memory released using the `free` function was not previously allocated using `malloc`, `calloc`, or `realloc`.

Risk

The `free` function releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.

The issue can highlight coding errors. For instance, you perhaps wanted to use the `free` function or a previous `malloc` function on a different pointer.

Fix

In most cases, you can fix the issue by removing the `free` statement. If the pointer is not allocated memory from the heap with `malloc` or `calloc`, you do not need to free the pointer. You can simply reuse the pointer as required.

If the issue highlights a coding error such as use of `free` or `malloc` on the wrong pointer, correct the error.

If the issue occurs because you use the `free` function to free memory allocated with the `new` operator, replace the `free` function with the `delete` operator.

Examples

Invalid Free of Pointer Error

```
#include <stdlib.h>

void Assign_Ones(void)
{
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;

  free(p);
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer `p` is deallocated using the `free` function. However, `p` points to a memory location that was not dynamically allocated.

Correction — Remove Pointer Deallocation

If the number of elements of the array `p` is known at compile time, one possible correction is to remove the deallocation of the pointer `p`.

```
#include <stdlib.h>

void Assign_Ones(void)
 {
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;
  /* Fix: Remove deallocation of p */
 }
```

Correction — Introduce Pointer Allocation

If the number of elements of the array `p` is not known at compile time, one possible correction is to dynamically allocate memory to the array `p`.

```
#include <stdlib.h>

void Assign_Ones(int num)
{
  int *p;
  /* Fix: Allocate memory dynamically to p */
  p=(int*) calloc(10,sizeof(int));
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
}
```

Check Information

Group: Dynamic Memory
Language: C | C++
Default: On
Command-Line Syntax: `BAD_FREE`
Impact: High
CWE ID: 404, 590, 762

See Also

`Find defects (-checkers)` | `Invalid deletion of pointer`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Memory leak

Memory allocated dynamically not freed

Description

This defect occurs when you do not free a block of memory allocated through `malloc`, `calloc`, `realloc`, or `new`. If the memory is allocated in a function, the defect does not occur if:

- Within the function, you free the memory using `free` or `delete`.
- The function returns the pointer assigned by `malloc`, `calloc`, `realloc`, or `new`.
- The function stores the pointer in a global variable or in a parameter.

Risk

Dynamic memory allocation functions such as `malloc` allocate memory on the heap. If you do not release the memory after use, you reduce the amount of memory available for another allocation. On embedded systems with limited memory, you might end up exhausting available heap memory even during program execution.

Fix

Determine the scope where the dynamically allocated memory is accessed. Free the memory block at the end of this scope.

To free a block of memory, use the `free` function on the pointer that was used during memory allocation. For instance:

```
ptr = (int*)malloc(sizeof(int));
...
free(ptr);
```

It is a good practice to allocate and free memory in the same module at the same level of abstraction. For instance, in this example, `func` allocates and frees memory at the same level but `func2` does not.

```
void func() {
  ptr = (int*)malloc(sizeof(int));
  {
    ...
  }
  free(ptr);
}

void func2() {
  {
   ptr = (int*)malloc(sizeof(int));
   ...
  }
  free(ptr);
}
```

See CERT-C Rule MEM00-C.

Examples

Dynamic Memory Not Released Before End of Function

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }


    *pi = 42;
    /* Defect: pi is not freed */
}
```

In this example, `pi` is dynamically allocated by `malloc`. The function `assign_memory` does not free the memory, nor does it return `pi`.

Correction — Free Memory

One possible correction is to free the memory referenced by `pi` using the `free` function. The `free` function must be called before the function `assign_memory` terminates

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }
    *pi = 42;

    /* Fix: Free the pointer pi*/
    free(pi);
}
```

Correction — Return Pointer from Dynamic Allocation

Another possible correction is to return the pointer `pi`. Returning `pi` allows the function calling `assign_memory` to free the memory block using `pi`.

```
#include<stdlib.h>
#include<stdio.h>

int* assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
```

```
        {
            printf("Memory allocation failed");
            return(pi);
        }
    *pi = 42;

    /* Fix: Return the pointer pi*/
    return(pi);
}
```

Memory Leak with New/Delete

```
#define NULL '\0'

void initialize_arr1(void)
{
    int *p_scalar = new int(5);
}

void initialize_arr2(void)
{
    int *p_array = new int[5];
}
```

In this example, the functions create two variables, `p_scalar` and `p_array`, using the `new` keyword. However, the functions end without cleaning up the memory for these pointers. Because the functions used `new` to create these variables, you must clean up their memory by calling `delete` at the end of each function.

Correction — Add Delete

To correct this error, add a `delete` statement for every `new` initialization. If you used brackets `[]` to instantiate a variable, you must call delete with brackets as well.

```
#define NULL '\0'

void initialize_arrs(void)
{
    int *p_scalar = new int(5);
    int *p_array = new int[5];

    delete p_scalar;
    p_scalar = NULL;

    delete[] p_array;
    p_scalar = NULL;
}
```

Check Information

Group: Dynamic memory
Language: C | C++
Default: Off
Command-Line Syntax: `MEM_LEAK`
Impact: Medium

CWE ID: 401, 404

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Mismatched alloc/dealloc functions on Windows

Improper deallocation function causes memory corruption issues

Description

This defect occurs when you use a Windows deallocation function that is not properly paired to its corresponding allocation function.

Risk

Deallocating memory with a function that does not match the allocation function can cause memory corruption or undefined behavior. If you are using an older version of Windows, the improper function can also cause compatibility issues with newer versions.

Fix

Properly pair your allocation and deallocation functions according to the functions listed in this table.

Allocation Function	Deallocation Function
`malloc()`	`free()`
`realloc()`	`free()`
`calloc()`	`free()`
`_aligned_malloc()`	`_aligned_free()`
`_aligned_offset_malloc()`	`_aligned_free()`
`_aligned_realloc()`	`_aligned_free()`
`_aligned_offset_realloc()`	`_aligned_free()`
`_aligned_recalloc()`	`_aligned_free()`
`_aligned_offset_recalloc()`	`_aligned_free()`
`_malloca()`	`_freea()`
`LocalAlloc()`	`LocalFree()`
`LocalReAlloc()`	`LocalFree()`
`GlobalAlloc()`	`GlobalFree()`
`GlobalReAlloc()`	`GlobalFree()`
`VirtualAlloc()`	`VirtualFree()`
`VirtualAllocEx()`	`VirtualFreeEx()`
`VirtualAllocExNuma()`	`VirtualFreeEx()`
`HeapAlloc()`	`HeapFree()`
`HeapReAlloc()`	`HeapFree()`

Examples

Memory Deallocated with Incorrect Function

```
#ifdef _WIN32_
#include <windows.h>
#else
#define _WIN32_
typedef void *HANDLE;
typedef HANDLE HGLOBAL;
typedef HANDLE HLOCAL;
typedef unsigned int UINT;
extern HLOCAL LocalAlloc(UINT uFlags, UINT uBytes);
extern HLOCAL LocalFree(HLOCAL hMem);
extern HGLOBAL GlobalFree(HGLOBAL hMem);
#endif

#define SIZE9 9


void func(void)
{
    /* Memory allocation */
    HLOCAL p = LocalAlloc(0x0000, SIZE9);

    if (p) {
        /* Memory deallocation. */
        GlobalFree(p);

    }
}
```

In this example, memory is allocated with `LocallAlloc()`. The program then erroneously uses `GlobalFree()` to deallocate the memory.

Correction — Properly Pair Windows Allocation and Deallocation Functions

When you allocate memory with `LocalAllocate()`, use `LocalFree()` to deallocate the memory.

```
#ifdef _WIN32_
#include <windows.h>
#else
#define _WIN32_
typedef void *HANDLE;
typedef HANDLE HGLOBAL;
typedef HANDLE HLOCAL;
typedef unsigned int UINT;
extern HLOCAL LocalAlloc(UINT uFlags, UINT uBytes);
extern HLOCAL LocalFree(HLOCAL hMem);
extern HGLOBAL GlobalFree(HGLOBAL hMem);
#endif

#define SIZE9 9
void func(void)
{
    /* Memory allocation */
```

```
    HLOCAL p = LocalAlloc(0x0000, SIZE9);
    if (p) {
        /* Memory deallocation. */
        LocalFree(p);
    }
}
```

Result Information

Group: Dynamic memory
Language: C | C++
Default: Off
Command-Line Syntax: `WIN_MISMATCH_DEALLOC`
Impact: Low
CWE ID: 404, 762

See Also

`Find defects (-checkers)` | `Invalid deletion of pointer` | `Invalid free of pointer`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Unprotected dynamic memory allocation

Pointer returned from dynamic allocation not checked for NULL value

Description

This defect occurs when you access dynamically allocated memory without first checking if the prior memory allocation succeeded.

Risk

When memory is dynamically allocated using `malloc`, `calloc`, or `realloc`, it returns a value `NULL` if the requested memory is not available. If the code following the allocation accesses the memory block without checking for this `NULL` value, this access is not protected from failures.

Fix

Check the return value of `malloc`, `calloc`, or `realloc` for NULL before accessing the allocated memory location.

```
int *ptr = malloc(size * sizeof(int));

if(ptr) /* Check for NULL */
{
   /* Memory access through ptr */
}
```

Examples

Unprotected dynamic memory allocation error

```
#include <stdlib.h>

void Assign_Value(void)
{
  int* p = (int*)calloc(5, sizeof(int));

  *p = 2;
  /* Defect: p is not checked for NULL value */

  free(p);
}
```

If the memory allocation fails, the function `calloc` returns `NULL` to `p`. Before accessing the memory through `p`, the code does not check whether `p` is `NULL`

Correction — Check for NULL Value

One possible correction is to check whether `p` has value `NULL` before dereference.

```
#include <stdlib.h>

void Assign_Value(void)
 {
```

```
   int* p = (int*)calloc(5, sizeof(int));

   /* Fix: Check if p is NULL */
   if(p!=NULL) *p = 2;

   free(p);
 }
```

Unprotected dynamic memory allocation error only on dereference

```
#include <stdlib.h>

struct recordType {
    const char* id;
    const char* data;
};

struct recordType *recordType_new(
    const char *id,
    unsigned int size)
{
    struct recordType *rec = calloc(1, sizeof(struct recordType));
    rec->id = strdup(id);

    const char *newData = calloc(1, size);
    rec->data = newData;
    return rec;
}
```

In this example, the checker raises a defect when you dereference the pointer `rec` without checking for a `NULL` value from the prior dynamic memory allocation.

A similar issue happens with the pointer `newData`. However, a defect is not raised because the pointer is not dereferenced but simply copied over to `rec->data`. Simply copying over a possibly null pointer is not an issue by itself. For instance, callers of the `recordType_new` function might check for `NULL` value of `rec->data` before dereferencing, thereby avoiding a null pointer dereference.

Check Information

Group: Dynamic memory
Language: C | C++
Default: Off
Command-Line Syntax: `UNPROTECTED_MEMORY_ALLOCATION`
Impact: Low
CWE ID: 253, 690, 789

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Use of previously freed pointer

Memory accessed after deallocation

Description

This defect occurs when you access a block of memory after freeing the block using the `free` function.

Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to access this block of memory can result in unpredictable behavior or even a segmentation fault.

Fix

The fix depends on the root cause of the defect. See if you intended to free the memory later or allocate another memory block to the pointer before access.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before dereferencing pointers, check them for NULL values and handle the error. In this way, you are protected against accessing a freed block.

Examples

Use of Previously Freed Pointer Error

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   {
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);

    j = *pi + shift;
    /* Defect: Reading a freed pointer */

    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, dereferencing`pi` after the `free` statement is not valid.

Correction — Free Pointer After Use

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
#include <stdlib.h>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

Check Information

Group: Dynamic memory
Language: C | C++
Default: On
Command-Line Syntax: FREED_PTR
Impact: High
CWE ID: 416, 825

See Also

Deallocation of previously deallocated pointer | Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Programming Defects

Abnormal termination of exit handler

Exit handler function interrupts the normal execution of a program

Description

This defect occurs when an exit handler itself calls another function that interrupts the program's expected termination sequence and causes an abnormal exit.

- Exit handlers are functions designated for execution when a program terminates. These functions are first registered with specific functions such as `atexit`, (WinAPI) `_onexit`, or `at_quick_exit()`.
- Some functions that can cause abnormal exits are `exit`, `abort`, `longjmp`, or (WinAPI) `_onexit`.

Risk

If your exit handler terminates your program, you can have undefined behavior. Abnormal program termination means other exit handlers are not invoked. These additional exit handlers may do additional clean up or other required termination steps.

Fix

In inside exit handlers, remove calls to functions that prevent the exit handler from terminating normally.

Examples

Exit Handler With Call to `exit`

```
#include <stdlib.h>

volatile int some_condition = 1;
void demo_exit1(void)
{
    /* ... Cleanup code ... */
    return;
}
void exitabnormalhandler(void)
{
    if (some_condition)
    {
        /* Clean up */
        exit(0);
    }
    return;
}

int demo_install_exitabnormalhandler(void)
{

    if (atexit(demo_exit1) != 0) /* demo_exit1() performs additional cleanup */
    {
        /* Handle error */
```

```
    }
    if (atexit(exitabnormalhandler) != 0)
    {
        /* Handle error */
    }
    /* ... Program code ... */
    return 0;
}
```

In this example, `demo_install_exitabnormalhandler` registers two exit handlers, `demo_exit1` and `exitabnormalhandler`. Exit handlers are invoked in the reverse order of which they are registered. When the program ends, `exitabnormalhandler` runs, then `demo_exit1`. However, `exitabnormalhandler` calls `exit` interrupting the program exit process. Having this `exit` inside an exit handler causes undefined behavior because the program is not finished cleaning up safely.

Correction — Remove `exit` from Exit Handler

One possible correction is to let your exit handlers terminate normally. For this example, `exit` is removed from `exitabnormalhandler`, allowing the exit termination process to complete as expected.

```
#include <stdlib.h>

volatile int some_condition = 1;
void demo_exit1(void)
{
    /* ... Cleanup code ... */
    return;
}
void exitabnormalhandler(void)
{
    if (some_condition)
    {
        /* Clean up */
        /* Return normally */
    }
    return;
}

int demo_install_exitabnormalhandler(void)
{

    if (atexit(demo_exit1) != 0) /* demo_exit1() continues clean up */
    {
        /* Handle error */
    }
    if (atexit(exitabnormalhandler) != 0)
    {
        /* Handle error */
    }
    /* ... Program code ... */
    return 0;
}
```

Result Information

Group: Programming

Language: C | C++
Default: Off
Command-Line Syntax: `EXIT_ABNORMAL_HANDLER`
Impact: Medium
CWE ID: 705

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2016b

Accessing object with temporary lifetime

Read or write operations on the object are undefined behavior

Description

This defect occurs when you attempt to read from or write to an object with temporary lifetime that is returned by a function call. In a structure or union returned by a function, and containing an array, the array members are temporary objects. The lifetime of temporary objects ends:

- When the full expression or full declarator containing the call ends, as defined in the C11 Standard.
- After the next sequence point, as defined in the C90 and C99 Standards. A sequence point is a point in the execution of a program where all previous evaluations are complete and no subsequent evaluation has started yet.

For C++ code, **Accessing object with temporary lifetime** raises a defect only when you write to an object with a temporary lifetime.

If the temporary lifetime object is returned by address, no defect is raised.

Risk

Modifying objects with temporary lifetime is undefined behavior and can cause abnormal program termination and portability issues.

Fix

Assign the object returned from the function call to a local variable. The content of the temporary lifetime object is copied to the variable. You can now modify it safely.

Examples

Modifying Temporary Lifetime Object Returned by Function Call

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6

struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

/* func_temp() returns a struct value containing
* an array with a temporary lifetime.
*/
```

```
int func(void) {

/*Writing to temporary lifetime object is
 undefined behavior
 */
    return ++(func_temp().a[0]);
}

void main(void) {
    (void)func();
}
```

In this example, `func_temp()` returns by value a structure with an array member `a`. This member has temporary lifetime. Incrementing it is undefined behavior.

Correction — Assign Returned Value to Local Variable Before Writing

One possible correction is to assign the return of the call to `func_temp()` to a local variable. The content of the temporary object `a` is copied to the variable, which you can safely increment.

```
 #include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6

struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

int func(void) {

/* Assign object returned by function call to
 *local variable
 */
    struct S_Array s = func_temp();

/* Local variable can safely be
 *incremented
 */
    ++(s.a[0]);
    return s.a[0];
}

void main(void) {
    (void)func();
}
```

Result Information

Group: Programming
Language: C | C++

Default: On
Command-Line Syntax: TEMP_OBJECT_ACCESS
Impact: Low
CWE ID: 825

See Also

`Find defects (-checkers)|Large pass-by-value argument|Misuse of structure with flexible array member|Write without a further read`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Alternating input and output from a stream without flush or positioning call

Undefined behavior for input or output stream operations

Description

This defect occurs when:

- You do not perform a flush or function positioning call between an output operation and a following input operation on a file stream in update mode.
- You do not perform a function positioning call between an input operation and a following output operation on a file stream in update mode.

Risk

Alternating input and output operations on a stream without an intervening flush or positioning call is undefined behavior.

Fix

Call `fflush()` or a file positioning function such as `fseek()` or `fsetpos()` between output and input operations on an update stream.

Call a file positioning function between input and output operations on an update stream.

Examples

Read After Write Without Intervening Flush

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";

void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;

    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
      {
```

```
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Read operation after write without
    intervening flush. */
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
          (void)fclose(file);
          /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

In this example, the file `demo.txt` is opened for reading and appending. After the call to `fwrite()`, a call to `fread()` without an intervening flush operation is undefined behavior.

Correction — Call `fflush()` Before the Read Operation

After writing data to the file, before calling `fread()`, perform a flush call.

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";


void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;

    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Buffer flush after write and before read */
    if (fflush(file) != 0)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
```

```
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `IO_INTERLEAVING`
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Assertion

Failed assertion statement

Description

This defect occurs when you use an `assert`, and the asserted expression is or could be false.

Note Polyspace does not flag `assert(0)` as an assertion defect because these statements are commonly used to disable certain sections of code.

Risk

Typically you use `assert` statements for functional testing in debug mode. An assertion failure found using static analysis indicates that the corresponding functional test would fail at run time.

Fix

The fix depends on the root cause of the defect. For instance, the root cause can be unconstrained input from an external source that eventually led to the assertion failure.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Check Assertion on Unsigned Integer

```
#include <assert.h>

void asserting_x(unsigned int theta) {
    theta =+ 5;
    assert(theta < 0);
}
```

In this example, the `assert` function checks if the input variable, `theta`, is less than or equal to zero. The assertion fails because `theta` is an unsigned integer, so the value at the beginning of the function is at least zero. The `+=` statement increases this positive value by five. Therefore, the range of `theta` is `[5..MAX_INT]`. `theta` is always greater than zero.

Correction — Change Assert Expression

One possible correction is to change the assertion expression. By changing the *less-than-or-equal-to* sign to a *greater-than-or-equal-to* sign, the assertion does not fail.

```
#include <assert.h>

void asserting_x(unsigned int theta) {
    theta =+ 5;
    assert(theta > 0);
}
```

Correction — Fix Code

One possible correction is to fix the code related to the assertion expression. If the assertion expression is true, fix your code so the assertion passes.

```
#include <assert.h>
#include <stdlib.h>

void asserting_x(int theta) {
    theta = -abs(theta);
    assert(theta < 0);
}
```

Asserting Zero

```
#include <assert.h>

#define FLAG 0

int main(void){
    int i_test_z = 0;
    float f_test_z = (float)i_test_z;

    assert(i_test_z);
    assert(f_test_z);
    assert(FLAG);

    return 0;
}
```

In this example, Polyspace does not flag `assert(FLAG)` as a violation because a macro defines `FLAG` as `0`. The Polyspace Bug Finder assertion checker does not flag assertions with a constant zero parameter, `assert(0)`. These types of assertions are commonly used as dynamic checks during runtime. By inserting `assert(0)`, you indicate that the program must not reach this statement during run time, otherwise the program crashes.

However, the assertion checker does flag failed assertions caused by a variable value equal to zero, as seen in the example with `assert(i_test_z)` and `assert(f_test_z)`.

Check Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: ASSERT
Impact: High

See Also

`Find defects (-checkers)`

Topics
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Bad file access mode or status

Access mode argument of function in `fopen` or `open` group is invalid

Description

This defect occurs when you use functions in the `fopen` or `open` group with invalid or incompatible file access modes, file creation flags, or file status flags as arguments. For instance, for the `open` function, examples of valid:

- Access modes include `O_RDONLY`, `O_WRONLY`, and `O_RDWR`
- File creation flags include `O_CREAT`, `O_EXCL`, `O_NOCTTY`, and `O_TRUNC`.
- File status flags include `O_APPEND`, `O_ASYNC`, `O_CLOEXEC`, `O_DIRECT`, `O_DIRECTORY`, `O_LARGEFILE`, `O_NOATIME`, `O_NOFOLLOW`, `O_NONBLOCK`, `O_NDELAY`, `O_SHLOCK`, `O_EXLOCK`, `O_FSYNC`, `O_SYNC` and so on.

The defect can occur in the following situations.

Situation	Risk	Fix
You pass an empty or invalid access mode to the `fopen` function. According to the ANSI C standard, the valid access modes for `fopen` are: • `r`,`r+` • `w`,`w+` • `a`,`a+` • `rb`, `wb`, `ab` • `r+b`, `w+b`, `a+b` • `rb+`, `wb+`, `ab+`	`fopen` has undefined behavior for invalid access modes. Some implementations allow extension of the access mode such as: • GNU: `rb+cmxe,ccs=utf` • Visual C++: `a+t`, where `t` specifies a text mode. However, your access mode string must begin with one of the valid sequences.	Pass a valid access mode to `fopen`.
You pass the status flag `O_APPEND` to the `open` function without combining it with either `O_WRONLY` or `O_RDWR`.	`O_APPEND` indicates that you intend to add new content at the end of a file. However, without `O_WRONLY` or `O_RDWR`, you cannot write to the file. The `open` function does not return -1 for this logical error.	Pass either `O_APPEND|O_WRONLY` or `O_APPEND|O_RDWR` as access mode.

Situation	Risk	Fix
You pass the status flags `O_APPEND` and `O_TRUNC` together to the `open` function.	`O_APPEND` indicates that you intend to add new content at the end of a file. However, `O_TRUNC` indicates that you intend to truncate the file to zero. Therefore, the two modes cannot operate together. The `open` function does not return -1 for this logical error.	Depending on what you intend to do, pass one of the two modes.
You pass the status flag `O_ASYNC` to the `open` function.	On certain implementations, the mode `O_ASYNC` does not enable signal-driven I/O operations.	Use the `fcntl(pathname, F_SETFL, O_ASYNC);` instead.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Invalid Access Mode with `fopen`

```
#include <stdio.h>

void func(void) {
    FILE *file = fopen("data.txt", "rw");
    if(file!=NULL) {
        fputs("new data",file);
        fclose(file);
    }
}
```

In this example, the access mode `rw` is invalid. Because `r` indicates that you open the file for reading and `w` indicates that you create a new file for writing, the two access modes are incompatible.

Correction — Use Either `r` or `w` as Access Mode

One possible correction is to use the access mode corresponding to what you intend to do.

```
#include <stdio.h>

void func(void) {
    FILE *file = fopen("data.txt", "w");
    if(file!=NULL) {
        fputs("new data",file);
```

```
        fclose(file);
    }
}
```

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: `BAD_FILE_ACCESS_MODE_STATUS`
Impact: Medium
CWE ID: 628, 686

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Call through non-prototyped function pointer

Function pointer declared without its type or number of parameters causes unexpected behavior

Description

This defect occurs when a function without a complete prototype is called using a function pointer.

A function prototype specifies the type and number of parameters.

Risk

Arguments passed to a function without a prototype might not match the number and type of parameters of the function definition, which can cause undefined behavior. If the parameters are restricted to a subset of their type domain, arguments from untrusted sources can trigger vulnerabilities in the called function.

Fix

Before calling the function through a pointer, provide a function prototype.

Examples

Argument Does Not Match Parameter Restriction

```
#include <stdio.h>
#include <limits.h>
#define SIZE2 2

typedef void (*func_ptr)();
extern int getchar_wrapper(void);
extern void restricted_int_sink(int i);
/* Integer value restricted to
range [-1, 255] */
extern void restricted_float_sink(double i);
/* Double value restricted to > 0.0 */



func_ptr generic_callback[SIZE2] =
{
    (func_ptr)restricted_int_sink,
    (func_ptr)restricted_float_sink
};

void func(void)
{
    int ic;
    ic = getchar_wrapper();
    /* Wrong index used for generic_callback.
    Negative 'int' passed to restricted_float_sink. */
    (*generic_callback[1])(ic);
}
```

In this example, a call through `func_ptr` passes `ic` as an argument to function `generic_callback[1]`. The type of `ic` can have negative values, while the parameter of `generic_callback[1]` is restricted to float values greater than `0.0`. Typically, compilers and static analysis tools cannot perform type checking when you do not provide a pointer prototype.

Correction — Provide Prototype of Pointer to Function

Pass the argument `ic` to a function with a parameter of type `int`, by using a properly prototyped pointer.

```
#include <stdio.h>
#include <limits.h>
#define SIZE2 2

typedef void (*func_ptr_proto)(int);
extern int getchar_wrapper(void);
extern void restricted_int_sink(int i);
/* Integer value restricted to
range [-1, 255] */
extern void restricted_float_sink(double i);
/* Double value restricted to > 0.0 */

func_ptr_proto generic_callback[SIZE2] =
{
    (func_ptr_proto)restricted_int_sink,
    (func_ptr_proto)restricted_float_sink
};

void func(void)
{
    int ic;
    ic = getchar_wrapper();
    /* ic passed to function through
properly prototyped pointer. */
    (*generic_callback[0])(ic);
}
```

Result Information

Group: Programming
Language: C
Default: On
Command-Line Syntax: `UNPROTOTYPED_FUNC_CALL`
Impact: Medium

See Also

`Declaration mismatch` | `Find defects (-checkers)` | `Unreliable cast of function pointer`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Call to memset with unintended value

`memset` or `wmemset` used with possibly incorrect arguments

Description

This defect occurs when Polyspace Bug Finder detects a use of the `memset` or `wmemset` function with possibly incorrect arguments.

`void *memset (void *ptr, int value, size_t num)` fills the first `num` bytes of the memory block that `ptr` points to with the specified `value`. If the argument `value` is incorrect, the memory block is initialized with an unintended value.

The unintended initialization can occur in the following cases.

Issue	Risk	Possible Fix
The second argument is `'0'` instead of `0` or `'\0'`.	The ASCII value of character `'0'` is `48` (decimal), `0x30` (hexadecimal), `069` (octal) but not `0` (or `'\0'`) .	If you want to initialize with `'0'`, use one of the ASCII values. Otherwise, use `0` or `'\0'`.
The second and third arguments are probably reversed. For instance, the third argument is a literal and the second argument is not a literal.	If the order is reversed, a memory block of unintended size is initialized with incorrect arguments.	Reverse the order of the arguments.
The second argument cannot be represented in a byte.	If the second argument cannot be represented in a byte, and you expect each byte of a memory block to be filled with that argument, the initialization does not occur as intended.	Apply a bit mask to the argument to produce a wrapped or truncated result that can be represented in a byte. When you apply a bit mask, make sure that it produces an expected result. For instance, replace `memset(a, -13, sizeof(a))` with `memset(a, (-13) & 0xFF, sizeof(a))`.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Value Cannot Be Represented in a Byte

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE];
    int c = -2;
    memset(buf, (char)c, sizeof(buf));
}
```

In this example, `(char)c` cannot be represented in a byte.

Correction — Apply Cast

One possible correction is to apply a cast so that the result can be represented in a byte. However, check that the result of the cast is an acceptable initialization value.

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE   ];
    int c = -2;
    memset(buf, (unsigned char)c, sizeof(buf));
}
```

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: `MEMSET_INVALID_VALUE`
Impact: Low
CWE ID: 665, 683

See Also

`Find defects (-checkers)` | `Use of memset with size argument zero`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Character value absorbed into EOF

Data type conversion makes a valid character value same as End-of-File (EOF)

Description

This defect occurs when you perform a data type conversion that makes a valid character value indistinguishable from `EOF` (End-of-File). Bug Finder flags the defect in one of the following situations:

- *End-of-File*: You perform a data type conversion such as from `int` to `char` that converts a non-EOF character value into `EOF`.

  ```
  char ch = (char)getchar()
  ```

  You then compare the result with EOF.

  ```
  if((int)ch == EOF)
  ```

  The conversion can be explicit or implicit.
- *Wide End-of-File*: You perform a data type conversion that can convert a non-WEOF wide character value into WEOF, and then compare the result with WEOF.

Risk

The data type `char` cannot hold the value `EOF` that indicates the end of a file. Functions such as `getchar` have return type `int` to accommodate `EOF`. If you convert from `int` to `char`, the values `UCHAR_MAX` (a valid character value) and `EOF` get converted to the same value -1 and become indistinguishable from each other. When you compare the result of this conversion with `EOF`, the comparison can lead to false detection of `EOF`. This rationale also applies to wide character values and `WEOF`.

Fix

Perform the comparison with `EOF` or `WEOF` before conversion.

Examples

Return Value of `getchar` Converted to `char`

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

char func(void)
{
    char ch;
    ch = getchar();
    if (EOF == (int)ch) {
        fatal_error();
    }
    return ch;
}
```

In this example, the return value of `getchar` is implicitly converted to `char`. If `getchar` returns `UCHAR_MAX`, it is converted to -1, which is indistinguishable from EOF. When you compare with EOF later, it can lead to a false positive.

Correction — Perform Comparison with EOF Before Conversion

One possible correction is to first perform the comparison with EOF, and then convert from `int` to `char`.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

char func(void)
{
    int i;
    i = getchar();
    if (EOF == i) {
        fatal_error();
    }
    else {
        return (char)i;
    }
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `CHAR_EOF_CONFUSED`
Impact: High
CWE ID: 704

See Also

`Errno not checked` | `Find defects (-checkers)` | `Invalid use of standard library integer routine` | `Misuse of sign-extended character value` | `Returned value of a sensitive function not checked`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Copy of overlapping memory

Source and destination arguments of a copy function have overlapping memory

Description

This defect occurs when there is a memory overlap between the source and destination argument of a copy function such as `memcpy` or `strcpy`. For instance, the source and destination arguments of `strcpy` are pointers to different elements in the same string.

Risk

If there is memory overlap between the source and destination arguments of copy functions, according to C standards, the behavior is undefined.

Fix

Determine if the memory overlap is what you want. If so, find an alternative function. For instance:

- If you are using `memcpy` to copy values from one memory location to another, use `memmove` instead of `memcpy`.
- If you are using `strcpy` to copy one string to another, use `memmove` instead of `strcpy`, as follows:

  ```
  s = strlen(source);
  memmove(destination, source, s + 1);
  ```

  `strlen` determines the string length without the null terminator. Therefore, you must move `s+1` bytes instead of `s` bytes.

Examples

Overlapping Copy

```
#include <string.h>

char str[] = {"ABCDEFGH"};

void my_copy() {
    strcpy(&str[0],(const char*)&str[2]);
}
```

In this example, because the source and destination argument are pointers to the same string `str`, there is memory overlap between their allowed buffers.

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: `OVERLAPPING_COPY`
Impact: Medium
CWE ID: 475, 628, 687

See Also

`Find defects (-checkers)` | `Overlapping assignment`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Declaration mismatch

Mismatch between function or variable declarations

Description

This defect occurs when a function or variable declaration does not match other instances of the function or variable.

Risk

When a mismatch occurs between two variable declarations in different compilation units, a typical linker follows an algorithm to pick one declaration for the variable. If you expect a variable declaration that is different from the one chosen by the linker, you can see unexpected results when the variable is used.

A similar issue can occur with mismatch in function declarations.

Fix

The fix depends on the type of declaration mismatch. If both declarations indeed refer to the same object, use the same declaration. If the declarations refer to different objects, change the names of the one of the variables. If you change a variable name, remember to make the change in all places that use the variable.

Sometimes, declaration mismatches can occur because the declarations are affected by previous preprocessing directives. For instance, a declaration occurs in a macro, and the macro is defined on one inclusion path but undefined in another. These declaration mismatches can be tricky to debug. Identify the divergence between the two inclusion paths and fix the conflicting macro definitions.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Inconsistent Declarations in Two Files

file1.c

```
int foo(void) {
    return 1;
}
```

file2.c

```
double foo(void);

int bar(void) {
    return (int)foo();
}
```

In this example, *file1.c* declares `foo()` as returning an integer. In *file2.c*, `foo()` is declared as returning a double. This difference raises a defect on the second instance of `foo` in *file2*.

Correction — Align the Function Return Values

One possible correction is to change the function declarations so that they match. In this example, by changing the declaration of `foo` in *file2.c* to match *file1.c*, the defect is fixed.

file1.c

```
int foo(void) {
    return 1;
}
```

file2.c

```
int foo(void);

int bar(void) {
    return foo();
}
```

Inconsistent Structure Alignment

test1.c

```
#include "square.h"
#include "circle.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

test2.c

```
#include "circle.h"
#include "square.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

circle.h

```
#pragma pack(1)

extern struct aCircle{
    int radius;
} circle;
```

square.h

```
extern struct aSquare {
    unsigned int side:1;
} square;
```

In this example, a declaration mismatch defect is raised on `square` in *square.h* because Polyspace infers that `square` in *square.h* does not have the same alignment as `square` in *test2.c*. This error occurs because the `#pragma pack(1)` statement in *circle.h* declares specific alignment. In *test2.c*, *circle.h* is included before *square.h*. Therefore, the `#pragma pack(1)` statement from *circle.h* is not reset to the default alignment after the `aCircle` structure. Because of this omission, *test2.c* infers that the `aSquare square` structure also has an alignment of `1` byte.

Correction — Close Packing Statements

One possible correction is to reset the structure alignment after the `aCircle` struct declaration. For the GNU or Microsoft Visual compilers, fix the defect by adding a `#pragma pack()` statement at the end of *circle.h*.

test1.c

```
#include "square.h"
#include "circle.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

test2.c

```
#include "circle.h"
#include "square.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

circle.h

```
#pragma pack(1)

extern struct aCircle{
    int radius;
} circle;

#pragma pack()
```

square.h

```
extern struct aSquare {
    unsigned int side:1;
} square;
```

Other compilers require different `#pragma pack` syntax. For your syntax, see the documentation for your compiler.

Correction — Use the `Ignore pragma pack directives` Option

One possible correction is to add the `Ignore pragma pack directives` option to your Bug Finder analysis. If you want the structure alignment to change for each structure, and you do not want to see this **Declaration mismatch** defect, use this correction.

1. On the Configuration pane, select the **Advanced Settings** pane.
2. In the **Other** box, enter `-ignore-pragma-pack`.
3. Rerun your analysis.

   The **Declaration mismatch** defect is resolved.

Check Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `DECL_MISMATCH`
Impact: High
CWE ID: 685, 686

See Also

`Find defects (-checkers)` | `Ignore pragma pack directives (-ignore-pragma-pack)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Environment pointer invalidated by previous operation

Call to `setenv` or `putenv` family function modifies environment pointed to by pointer

Description

This defect occurs when you use the third argument of *main()* in a hosted environment to access the environment after an operation modifies the environment. In a hosted environment, many C implementations support the nonstandard syntax:

```
main (int argc, char *argv[], char *envp[])
```

A call to a `setenv` or `putenv` family function modifies the environment pointed to by `*envp`.

Risk

When you modify the environment through a call to a `setenv` or `putenv` family function, the environment memory can potentially be reallocated. The hosted environment pointer is not updated and might point to an incorrect location. A call to this pointer can return unexpected results or cause an abnormal program termination.

Fix

Do not use the hosted environment pointer. Instead, use global external variable `environ` in Linux, `_environ` or `_wenviron` in Windows, or their equivalent. When you modify the environment, these variables are updated.

Examples

Access Environment Through Pointer envp

```
#include <stdio.h>
#include <stdlib.h>

extern int check_arguments(int argc, char **argv, char **envp);
extern void use_envp(char **envp);

/* envp is from main function */
int func(char **envp)
{
    /* Call to setenv may cause environment
     *memory to be reallocated
     */
    if (setenv(("MY_NEW_VAR"),("new_value"),1) != 0)
    {
        /* Handle error */
        return -1;
    }
    /* envp not updated after call to setenv, and may
     *point to incorrect location.
     **/
```

```
    if (envp != ((void *)0)) {
        use_envp(envp);
/* No defect on second access to
*envp because defect already raised */
    }
    return 0;
}

void  main(int argc, char **argv, char **envp)
{
    if (check_arguments(argc, argv, envp))
    {
        (void)func(envp);
    }
}
```

In this example, `envp` is accessed inside `func()` after a call to `setenv` that can reallocate the environment memory. `envp` can point to an incorrect location because it is not updated after `setenv` modifies the environment. No defect is raised when `use_envp()` is called because the defect is already raised on the previous line of code.

Correction — Use Global External Variable `environ`

One possible correction is to access the environment by using a variable that is always updated after a call to `setenv`. For instance, in the following code, the pointer `envp` is still available from `main()`, but the environment is accessed in `func()` through the global external variable `environ`.

```
#include <stdio.h>
#include <stdlib.h>
extern char **environ;

extern int check_arguments(int argc, char **argv, char **envp);
extern void use_envp(char **envp);

int func(void)
{
    if (setenv(("MY_NEW_VAR"), ("new_value"),1) != 0) {
        /* Handle error */
        return -1;
    }
  /* Use global external variable environ
   *which is always updated after a call to setenv */

    if (environ != NULL) {
        use_envp(environ);
    }
    return 0;
}

void  main(int argc, char **argv, char **envp)
{
    if (check_arguments(argc, argv, envp))
    {
        (void)func();
    }
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `INVALID_ENV_POINTER`
Impact: Medium
CWE ID: 825

See Also

`Find defects (-checkers)|Misuse of return value from nonreentrant standard function|Modification of internal buffer returned from nonreentrant standard function`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Errno not reset

`errno` not reset before calling a function that sets `errno`

Description

This defect occurs when you do not reset `errno` before calling a function that sets `errno` to indicate error conditions. However, you check `errno` for those error conditions after the function call.

Risk

The `errno` is not clean and can contain values from a previous call. Checking `errno` for errors can give the false impression that an error occurred.

`errno` is set to zero at program startup but subsequently, `errno` is not reset by a C standard library function. You must explicitly set `errno` to zero when required.

Fix

Before calling a function that sets `errno` to indicate error conditions, reset `errno` to zero explicitly.

Examples

`errno` Not Reset Before Call to `strtod`

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>

#define fatal_error() abort()

double func(const char *s1, const char *s2)
{
    double f1;
    f1 = strtod (s1, NULL);
    if (0 == errno) {
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
                  {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

In this example, `errno` is not reset to 0 before the first call to `strtod`. Checking `errno` for 0 later can lead to a false positive.

Correction — Reset `errno` Before Call

One possible correction is to reset `errno` to 0 before calling `strtod`.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>

#define fatal_error() abort()

double func(const char *s1, const char *s2)
{
    double f1;
    errno = 0;
    f1 = strtod (s1, NULL);
    if (0 == errno) {
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
              {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `MISSING_ERRNO_RESET`
Impact: High
CWE ID: 253, 456, 703

See Also

`Errno not checked` | `Find defects (-checkers)` | `Misuse of errno` | `Returned value of a sensitive function not checked`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Exception caught by value

`catch` statement accepts an object by value

Description

This defect occurs when a `catch` statement accepts an object by value.

Risk

If a `throw` statement passes an object and the corresponding `catch` statement accepts the exception by value, the object is copied to the `catch` statement parameter. This copy can lead to unexpected behavior such as:

- Object slicing, if the `throw` statement passes a derived class object.
- Undefined behavior of the exception, if the copy fails.

Fix

Catch the exception by reference or by pointer. Catching an exception by reference is recommended.

Examples

Standard Exception Caught by Value

```
#include <exception>

extern void print_str(const char* p);
extern void throw_exception();

void func() {
    try {
        throw_exception();
    }

    catch(std::exception exc) {
        print_str(exc.what());
    }
}
```

In this example, the `catch` statement takes a `std::exception` object by value. Catching an exception by value causes copying of the object. It can cause undefined behavior of the exception if the copy fails.

Correction: Catch Exception by Reference

One possible solution is to catch the exception by reference.

```
#include <exception>

extern void print_str(const char* p);
extern void throw_exception();

void corrected_excpcaughtbyvalue() {
```

```
    try {
        throw_exception();
    }
    catch(std::exception& exc) {
        print_str(exc.what());
    }
}
```

Derived Class Exception Caught by Value

```
#include <exception>
#include <string>
#include <typeinfo>
#include <iostream>

// Class declarations
class BaseExc {
public:
    explicit BaseExc();
    virtual ~BaseExc() {};
protected:
    BaseExc(const std::string& type);
private:
    std::string _id;
};

class IOExc: public BaseExc {
public:
    explicit IOExc();
};

//Class method declarations
BaseExc::BaseExc():_id(typeid(this).name()) {
}
BaseExc::BaseExc(const std::string& type): _id(type) {
}
IOExc::IOExc(): BaseExc(typeid(this).name()) {
}

int input(void);

int main(void) {
    int rnd = input();
    try {
        if (rnd==0) {
            throw IOExc();
        } else {
            throw BaseExc();
        }
    }


    catch(BaseExc exc) {
        std::cout << "Intercept BaseExc" << std::endl;
    }
    return 0;
}
```

In this example, the `catch` statement takes a `BaseExc` object by value. Catching exceptions by value causes copying of the object. The copying can cause:

- Undefined behavior of the exception if it fails.
- Object slicing if an exception of the derived class `IOExc` is caught.

Correction — Catch Exceptions by Reference

One possible correction is to catch exceptions by reference.

```
#include <exception>
#include <string>
#include <typeinfo>
#include <iostream>

// Class declarations
class BaseExc {
public:
    explicit BaseExc();
    virtual ~BaseExc() {};
protected:
    BaseExc(const std::string& type);
private:
    std::string _id;
};

class IOExc: public BaseExc {
public:
    explicit IOExc();
};

//Class method declarations
BaseExc::BaseExc():_id(typeid(this).name()) {
}
BaseExc::BaseExc(const std::string& type): _id(type) {
}
IOExc::IOExc(): BaseExc(typeid(this).name()) {
}

int input(void);

int main(void) {
    int rnd = input();
    try {
        if (rnd==0) {
            throw IOExc();
        } else {
            throw BaseExc();
        }
    }



    catch(BaseExc& exc) {
        std::cout << "Intercept BaseExc" << std::endl;
    }
    return 0;
}
```

Result Information

Group: Programming
Language: C++
Default: On
Command-Line Syntax: `EXCP_CAUGHT_BY_VALUE`
Impact: Medium

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Exception handler hidden by previous handler

`catch` statement is not reached because of an earlier `catch` statement for the same exception

Description

This defect occurs when a `catch` statement is not reached because a previous `catch` statement handles the exception.

For instance, a `catch` statement accepts an object of a class `my_exception` and a later `catch` statement accepts one of the following:

- An object of the `my_exception` class.
- An object of a class derived from the `my_exception` class.

Risk

Because the `catch` statement is not reached, it is effectively dead code.

Fix

One possible fix is to remove the redundant `catch` statement.

Another possible fix is to reverse the order of `catch` statements. Place the `catch` statement that accepts the derived class exception before the `catch` statement that accepts the base class exception.

Examples

`catch` Statement Hidden by Previous Statement

```
#include <new>

extern void print_str(const char* p);
extern void throw_exception();

void func() {
    try {
        throw_exception();
    }
    catch(std::exception& exc) {
        print_str(exc.what());
    }

    catch(std::bad_alloc& exc) {
        print_str(exc.what());
    }
}
```

In this example, the second `catch` statement accepts a `std::bad_alloc` object. Because the `std::bad_alloc` class is derived from a `std::exception` class, the second `catch` statement is hidden by the previous `catch` statement that accepts a `std::exception` object.

The defect appears on the parameter type of the `catch` statement. To find which `catch` statement hides the current `catch` statement:

1. On the **Source** pane, right-click the keyword `catch` and select **Search For `"catch"` in Current Source File**.
2. On the **Search** pane, click each search result, proceeding backwards from the current `catch` statement. Continue until you find the `catch` statement that hides the `catch` statement with the defect.

Correction — Reorder `catch` Statement

One possible correction is to place the `catch` statement with the derived class parameter first.

```
#include <new>

extern void print_str(const char* p);
extern void throw_exception();

void corrected_excphandlerhidden() {
    try {
        throw_exception();
    }

    catch(std::bad_alloc& exc) {
        print_str(exc.what());
    }
    catch(std::exception& exc) {
        print_str(exc.what());
    }
}
```

Result Information

Group: Programming
Language: C++
Default: On
Command-Line Syntax: `EXCP_HANDLER_HIDDEN`
Impact: Medium
CWE ID: 755

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Floating point comparison with equality operators

Imprecise comparison of floating-point variables

Description

This defect occurs when you use an equality (==) or inequality (!=) operation with floating-point numbers.

Polyspace does not raise a defect for an equality or inequality operation with floating-point numbers when:

- The comparison is between two float constants.

  ```
  float flt = 1.0;
  if (flt == 1.1)
  ```

- The comparison is between a constant and a variable that can take a finite, reasonably small number of values.

  ```
  float x;

  int rand = random();
  switch(rand) {
  case 1: x = 0.0; break;
  case 2: x = 1.3; break;
  case 3: x = 1.7; break;
  case 4: x = 2.0; break;
  default: x = 3.5; break; }
  …
  if (x==1.3)
  ```

- The comparison is between floating-point expressions that contain only integer values.

  ```
  float x = 0.0;
  for (x=0.0;x!=100.0;x+=1.0) {
  …
  if (random) break;
  }

  if (3*x+4==2*x-1)
  …
  if (3*x+4 == 1.3)
  ```

- One of the operands is `0.0`, unless you use the option flag `-detect-bad-float-op-on-zero`.

  ```
  /* Defect detected when
  you use the option flag */

  if (x==0.0f)
  ```

  If you are running an analysis through the user interface, you can enter this option in the **Other** field, under the **Advanced Settings** node on the **Configuration** pane. See `Other`.

  At the command line, add the flag to your analysis command.

```
polyspace-bug-finder -sources filename ^
-checkers BAD_FLOAT_OP -detect-bad-float-op-on-zero
```

Risk

Checking for equality or inequality of two floating-point values might return unexpected results because floating-point representations are inexact and involve rounding errors.

Fix

Instead of checking for equality of floating-point values:

```
if (val1 == val2)
```

check if their difference is less than a predefined tolerance value (for instance, the value `FLT_EPSILON` defined in `float.h`):

```
#include <float.h>
if(fabs(val1-val2) < FLT_EPSILON)
```

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Examples

Floats Inequality in `for`-loop

```
#include <stdio.h>
#include <math.h>
#include <float.h>

void func(void)
{
    float f;
    for (f = 1.0; f != 2.0; f = f + 0.1)
        (void)printf("Value: %f\n", f);
}
```

In this function, the `for`-loop tests the inequality of `f` and the number 2.0 as a stopping mechanism. The number of iterations is difficult to determine, or might be infinite, because of the imprecision in floating-point representation.

Correction — Change the Operator

One possible correction is to use a different operator that is not as strict. For example, an inequality like >= or <=.

```
#include <stdio.h>
#include <math.h>
#include <float.h>

void func(void)
{
    float f;
    for (f = 1.0; f <= 2.0; f = f + 0.1)
```

```
        (void)printf("Value: %f\n", f);
}
```

Check Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: BAD_FLOAT_OP
Impact: Medium
CWE ID: 873

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Format string specifiers and arguments mismatch

String specifiers do not match corresponding arguments

Description

This defect occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
  const char *message = "License not available";
  int err_number = ;-4
  printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Printing a Float

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", fst);
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {
```

```
    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

Check Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `STRING_FORMAT`
Impact: Low
CWE ID: 683, 685, 686

See Also

`Find defects (-checkers)` | `Invalid use of standard library string routine`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

External Websites

Standard library output functions

Introduced in R2013b

Function called from signal handler not asynchronous-safe

Call to interrupted function causes undefined program behavior

Description

This defect occurs when a signal handler calls a function that is not asynchronous-safe according to the POSIX standard. An asynchronous-safe function can be interrupted at any point in its execution, then called again without causing an inconsistent state. It can also correctly handle global data that might be in an inconsistent state.

If a signal handler calls another function that calls an asynchronous-unsafe function, the defect appears on the function call in the signal handler. The defect traceback shows the full path from the signal handler to the asynchronous-unsafe function.

Risk

When a signal handler is invoked, the execution of the program is interrupted. After the handler is finished, program execution resumes at the point of interruption. If a function is executing at the time of the interruption, calling it from within the signal handler is undefined behavior, unless it is asynchronous-safe.

Fix

The POSIX standard defines these functions as asynchronous-safe. You can call these functions from a signal handler.

`_exit()`	`getpgrp()`	`setsockopt()`
`_Exit()`	`getpid()`	`setuid()`
`abort()`	`getppid()`	`shutdown()`
`accept()`	`getsockname()`	`sigaction()`
`access()`	`getsockopt()`	`sigaddset()`
`aio_error()`	`getuid()`	`sigdelset()`
`aio_return()`	`kill()`	`sigemptyset()`
`aio_suspend()`	`link()`	`sigfillset()`
`alarm()`	`linkat()`	`sigismember()`
`bind()`	`listen()`	`signal()`
`cfgetispeed()`	`lseek()`	`sigpause()`
`cfgetospeed()`	`lstat()`	`sigpending()`
`cfsetispeed()`	`mkdir()`	`sigprocmask()`
`cfsetospeed()`	`mkdirat()`	`sigqueue()`
`chdir()`	`mkfifo()`	`sigset()`
`chmod()`	`mkfifoat()`	`sigsuspend()`

chown()	mknod()	sleep()
clock_gettime()	mknodat()	sockatmark()
close()	open()	socket()
connect()	openat()	socketpair()
creat()	pathconf()	stat()
dup()	pause()	symlink()
dup2()	pipe()	symlinkat()
execl()	poll()	sysconf()
execle()	posix_trace_event()	tcdrain()
execv()	pselect()	tcflow()
execve()	pthread_kill()	tcflush()
faccessat()	pthread_self()	tcgetattr()
fchdir()	pthread_sigmask()	tcgetpgrp()
fchmod()	quick_exit()	tcsendbreak()
fchmodat()	raise()	tcsetattr()
fchown()	read()	tcsetpgrp()
fchownat()	readlink()	time()
fcntl()	readlinkat()	timer_getoverrun()
fdatasync()	recv()	timer_gettime()
fexecve()	recvfrom()	timer_settime()
fork()	recvmsg()	times()
fpathconf()	rename()	umask()
fstat()	renameat()	uname()
fstatat()	rmdir()	unlink()
fsync()	select()	unlinkat()
ftruncate()	sem_post()	utime()
futimens()	send()	utimensat()
getegid()	sendmsg()	utimes()
geteuid()	sendto()	wait()
getgid()	setgid()	waitpid()
getgroups()	setpgid()	write()
getpeername()	setsid()	

Functions not in the previous table are not asynchronous-safe, and should not be called from a signal hander.

Examples

Call to `printf()` Inside Signal Handler

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

#define SIZE20 20

extern volatile sig_atomic_t e_flag;

void display_info(const char *info)
{
    if (info)
    {
        (void)fputs(info, stderr);
    }
}

void sig_handler(int signum)
{
    /* Call function printf() that is not
    asynchronous-safe */
    printf("signal %d received.", signum);
    e_flag = 1;
}

int main(void)
{
    e_flag = 0;
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info = (char *)calloc(SIZE20, sizeof(char));
    if (info == NULL)
    {
        /* Handle Error */
    }
    while (!e_flag)
    {
        /* Main loop program code */
        display_info(info);
        /* More program code */
    }
    free(info);
    info = NULL;
    return 0;
}
```

In this example, `sig_handler` calls `printf()` when catching a signal. If the handler catches another signal while `printf()` is executing, the behavior of the program is undefined.

Correction — Set Flag Only in Signal Handler

Use your signal handler to set only the value of a flag. `e_flag` is of type volatile `sig_atomic_t`. `sig_handler` can safely access it asynchronously.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

#define SIZE20 20

extern volatile sig_atomic_t e_flag;

void display_info(const char *info)
{
    if (info)
    {
        (void)fputs(info, stderr);
    }
}

void sig_handler1(int signum)
{
    int s0 = signum;
    e_flag = 1;
}

int func(void)
{
    e_flag = 0;
    if (signal(SIGINT, sig_handler1) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info = (char *)calloc(SIZE20, 1);
    if (info == NULL)
    {
        /* Handle error */
    }
    while (!e_flag)
    {
        /* Main loop program code */
        display_info(info);
        /* More program code */
    }
    free(info);
    info = NULL;
    return 0;
}
```

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: `SIG_HANDLER_ASYNC_UNSAFE`
Impact: Medium
CWE ID: 364, 387, 413, 479, 663, 828

See Also

`Find defects (-checkers)` | `Function called from signal handler not asynchronous-safe (strict)` | `Return from computational exception signal handler` | `Shared data access within signal handler` | `Signal call from within signal handler`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Function called from signal handler not asynchronous-safe (strict)

Call to interrupted function causes undefined program behavior

Description

This defect occurs when a signal handler calls a function that is not asynchronous-safe according to the C standard. An asynchronous-safe function can be interrupted at any point in its execution, then called again without causing an inconsistent state. It can also correctly handle global data that might be in an inconsistent state.

When you select the checker **Function called from signal handler not asynchronous-safe**, the checker detects calls to functions that are not asynchronous-safe according to the POSIX standard. **Function called from signal handler not asynchronous-safe (strict)** does not raise a defect for these cases. **Function called from signal handler not asynchronous-safe (strict)** raises a defect for functions that are asynchronous-safe according to the POSIX standard but not according to the C standard.

If a signal handler calls another function that calls an asynchronous-unsafe function, the defect appears on the function call in the signal handler. The defect traceback shows the full path from the signal handler to the asynchronous-unsafe function.

Risk

When a signal handler is invoked, the execution of the program is interrupted. After the handler is finished, program execution resumes at the point of interruption. If a function is executing at the time of the interruption, calling it from within the signal handler is undefined behavior, unless it is asynchronous-safe.

Fix

The C standard defines the following functions as asynchronous-safe. You can call these functions from a signal handler:

- `abort()`
- `_Exit()`
- `quick_exit()`
- `signal()`

Examples

Call to `raise()` Inside Signal Handler

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
```

```
#include <unistd.h>

void SIG_ERR_handler(int signum)
{
    int s0 = signum;
    /* SIGTERM specific handling */
}

void sig_handler(int signum)
{
    int s0 = signum;
    /* Call raise() */
    if (raise(SIGTERM) != 0) {
        /* Handle error */
    }
}

int finc(void)
{
    if (signal(SIGTERM, SIG_ERR_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return 0;
}
```

In this example, `sig_handler` calls `raise()` when catching a signal. If the handler catches another signal while `raise()` is executing, the behavior of the program is undefined.

Correction — Remove Call to `raise()` in Signal Handler

According to the C standard, the only functions that you can safely call from a signal handler are `abort()`, `_Exit()`, `quick_exit()`, and `signal()`.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

void SIG_ERR_handler(int signum)
{
    int s0 = signum;
    /* SIGTERM specific handling */
```

```
}
void sig_handler(int signum)
{
    int s0 = signum;


}

int func(void)
{
    if (signal(SIGTERM, SIG_ERR_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return 0;
}
```

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: `SIG_HANDLER_ASYNC_UNSAFE_STRICT`
Impact: Medium
CWE ID: 364, 387, 413, 479, 663, 828

See Also

`Find defects (-checkers)` | `Function called from signal handler not asynchronous-safe` | `Shared data access within signal handler` | `Signal call from within signal handler`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Improper array initialization

Incorrect array initialization when using initializers

Description

This defect occurs when Polyspace Bug Finder considers that an array initialization using initializers is incorrect.

This defect applies to normal and designated initializers. In C99, with designated initializers, you can place the elements of an array initializer in any order and implicitly initialize some array elements. The designated initializers use the array index to establish correspondence between an array element and an array initializer element. For instance, the statement `int arr[6] = { [4] = 29, [2] = 15 }` is equivalent to `int arr[6] = { 0, 0, 15, 0, 29, 0 }`.

You can use initializers incorrectly in one of the following ways.

Issue	Risk	Possible Fix
In your initializer for a one-dimensional array, you have more elements than the array size.	Unused array initializer elements indicate a possible coding error.	Increase the array size or remove excess elements.
You place the braces enclosing initializer values incorrectly.	Because of the incorrect placement of braces, some array initializer elements are not used. Unused array initializer elements indicate a possible coding error.	Place braces correctly.
In your designated initializer, you do not initialize the first element of the array explicitly.	The implicit initialization of the first array element indicates a possible coding error. You possibly overlooked the fact that array indexing starts from 0.	Initialize all elements explicitly.
In your designated initializer, you initialize an element twice.	The first initialization is overridden. The redundant first initialization indicates a possible coding error.	Remove the redundant initialization.
You use designated and nondesignated initializers in the same initialization.	You or another reviewer of your code cannot determine the size of the array by inspection.	Use either designated or nondesignated initializers.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do

not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Incorrectly Placed Braces (C Only)

```
int arr[2][3]
= {{1, 2},
    {3, 4},
    {5, 6}
};
```

In this example, the array `arr` is initialized as `{1,2,0,3,4,0}`. Because the initializer contains `{5,6}`, you might expect the array to be initialized `{1,2,3,4,5,6}`.

Correction — Place Braces Correctly

One possible correction is to place the braces correctly so that all elements are explicitly initialized.

```
int a1[2][3]
= {{1, 2, 3},
    {4, 5, 6}
};
```

First Element Not Explicitly Initialized

```
int arr[5]
= {
    [1] = 2,
    [2] = 3,
    [3] = 4,
    [4] = 5
};
```

In this example, `arr[0]` is not explicitly initialized. It is possible that the programmer did not consider that the array indexing starts from 0.

Correction — Explicitly Initialize All Elements

One possible correction is to initialize all elements explicitly.

```
int arr[5]
= {
    [0] = 1,
    [1] = 2,
    [2] = 3,
    [3] = 4,
    [4] = 5
};
```

Element Initialized Twice

```
int arr[5]
= {
    [0] = 1,
    [1] = 2,
    [2] = 3,
    [2] = 4,
    [4] = 5
};
```

In this example, `arr[2]` is initialized twice. The first initialization is overridden. In this case, because `arr[3]` was not explicitly initialized, it is possible that the programmer intended to initialize `arr[3]` when `arr[2]` was initialized a second time.

Correction — Fix Redundant Initialization

One possible correction is to eliminate the redundant initialization.

```
int arr[5]
= {
    [0] = 1,
    [1] = 2,
    [2] = 3,
    [3] = 4,
    [4] = 5
};
```

Mix of Designated and Nondesignated Initializers

```
int arr[]
= {
    [0] = 1,
    [3] = 3,
    4,
    [5] = 5,
    6
    };
```

In this example, because a mix of designated and nondesignated initializers are used, it is difficult to determine the size of `arr` by inspection.

Correction — Use Only Designated Initializers

One possible correction is to use only designated initializers for array initialization.

```
int arr[]
= {
    [0] = 1,
    [3] = 3,
    [4] = 4,
    [5] = 5,
    [6] = 6
};
```

Result Information

Group: Programming

Language: C | C++
Default: On
Command-Line Syntax: `IMPROPER_ARRAY_INIT`
Impact: Medium
CWE ID: 665

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Incorrect data type passed to va_arg

Data type of variadic function argument does not match type in `va_arg` call

Description

This defect occurs when the data type in a `va_arg` call does not match the data type of the variadic function argument that `va_arg` reads.

For instance, you pass an `unsigned char` argument to a variadic function `func`. Because of default argument promotion, the argument is promoted to `int`. When you use a `va_arg` call that reads an `unsigned char` argument, a type mismatch occurs.

```
void func (int n, ...) {
   ...
   va_list args;
   va_arg(args, unsigned char);
   ...
}

void main(void) {
   unsigned char c;
   func(1,c);
}
```

Risk

In a variadic function (function with variable number of arguments), you use `va_arg` to read each argument from the variable argument list (`va_list`). The `va_arg` use does not guarantee that there actually exists an argument to read or that the argument data type matches the data type in the `va_arg` call. You have to make sure that both conditions are true.

Reading an incorrect type with a `va_arg` call can result in undefined behavior. Because function arguments reside on the stack, you might access an unwanted area of the stack.

Fix

Make sure that the data type of the argument passed to the variadic function matches the data type in the `va_arg` call.

Arguments of a variadic function undergo default argument promotions. The argument data types of a variadic function cannot be determined from a prototype. The arguments of such functions undergo default argument promotions (see Sec. 6.5.2.2 and 7.15.1.1 in the C99 Standard). Integer arguments undergo integer promotion and arguments of type `float` are promoted to `double`. For integer arguments, if a data type can be represented by an `int`, for instance, `char` or `short`, it is promoted to an `int`. Otherwise, it is promoted to an `unsigned int`. All other arguments do not undergo promotion.

To avoid undefined and implementation-defined behavior, minimize the use of variadic functions. Use the checkers for `MISRA C:2012 Rule 17.1` or `MISRA C++:2008 Rule 8-4-1` to detect use of variadic functions.

Examples

`char` Used as Function Argument Type and `va_arg` argument

```
#include <stdarg.h>
#include <stdio.h>

unsigned char func(size_t count, ...) {
    va_list ap;
    unsigned char result = 0;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, unsigned char);
    }
    va_end(ap);
    return result;
}

void func_caller(void) {
    unsigned char c = 0x12;
    (void)func(1, c);
}
```

In this example, `func` takes an `unsigned char` argument, which undergoes default argument promotion to `int`. The data type in the `va_arg` call is still `unsigned char`, which does not match the `int` argument type.

Correction — Use `int` as `va_arg` Argument

One possible correction is to read an `int` argument with `va_arg`.

```
#include <stdarg.h>
#include <stdio.h>

unsigned char func(size_t count, ...) {
    va_list ap;
    unsigned char result = 0;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
    }
    va_end(ap);
    return result;
}

void func_caller(void) {
    unsigned char c = 0x12;
    (void)func(1, c);
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `VA_ARG_INCORRECT_TYPE`
Impact: Medium

CWE ID: 686

See Also

`Find defects (-checkers)` | `Invalid va_list argument` | `Too many va_arg calls for current argument list`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Incorrect pointer scaling

Implicit scaling in pointer arithmetic might be ignored

Description

This defect occurs when Polyspace Bug Finder considers that you are ignoring the implicit scaling in pointer arithmetic.

For instance, the defect can occur in the following situations.

Situation	Risk	Possible Fix
You use the `sizeof` operator in arithmetic operations on a pointer.	The `sizeof` operator returns the size of a data type in number of bytes. Pointer arithmetic is already implicitly scaled by the size of the data type of the pointed variable. Therefore, the use of `sizeof` in pointer arithmetic produces unintended results.	Do not use `sizeof` operator in pointer arithmetic.
You perform arithmetic operations on a pointer, and then apply a cast.	Pointer arithmetic is implicitly scaled. If you do not consider this implicit scaling, casting the result of a pointer arithmetic produces unintended results.	Apply the cast before the pointer arithmetic.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Use of `sizeof` Operator

```
void func(void) {
    int arr[5] = {1,2,3,4,5};
    int *ptr = arr;

    int value_in_position_2 = *(ptr + 2*(sizeof(int)));
}
```

In this example, the operation `2*(sizeof(int))` returns twice the size of an `int` variable in bytes. However, because pointer arithmetic is implicitly scaled, the number of bytes by which `ptr` is offset is `2*(sizeof(int))*(sizeof(int))`.

In this example, the incorrect scaling shifts `ptr` outside the bounds of the array. Therefore, a **Pointer access out of bounds** error appears on the * operation.

Correction — Remove `sizeof` Operator

One possible correction is to remove the `sizeof` operator.

```
void func(void) {
    int arr[5] = {1,2,3,4,5};
    int *ptr = arr;

    int value_in_position_2 = *(ptr + 2);
}
```

Cast Following Pointer Arithmetic

```
int func(void) {
    int x = 0;
    char r = *(char *)(&x + 1);
    return r;
}
```

In this example, the operation `&x + 1` offsets `&x` by `sizeof(int)`. Following the operation, the resulting pointer points outside the allowed buffer. When you dereference the pointer, a **Pointer access out of bounds** error appears on the * operation.

Correction — Apply Cast Before Pointer Arithmetic

If you want to access the second byte of `x`, first cast `&x` to a `char*` pointer and then perform the pointer arithmetic. The resulting pointer is offset by `sizeof(char)` bytes and still points within the allowed buffer, whose size is `sizeof(int)` bytes.

```
int func(void) {
    int x = 0;
    char r = *((char *)(&x )+ 1);
    return r;
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `BAD_PTR_SCALING`
Impact: Medium
CWE ID: 468

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”

“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Incorrect type data passed to va_start

Data type of second argument to `va_start` macro leads to undefined behavior

Description

This defect occurs when the second argument of the `va_start` macro has one of these data types:

- A data type that changes when undergoing default argument promotion.

  For instance, `char` and `short` undergo promotion to `int` or `unsigned int` and `float` undergoes promotion to `double`. The types `int` and `double` do not change under default argument promotion.
- (C only) A register type or a data type declared with the `register` qualifier.
- (C++ only) A reference data type.
- (C++ only) A data type that has a nontrivial copy constructor or a nontrivial move constructor.

Risk

In a variadic function or function with variable number of arguments:

```
void multipleArgumentFunction(int someArg, short rightmostFixedArg, ...) {
    va_list myList;
    va_start(myList, rightmostFixedArg);
    ...
    va_end(myList);
}
```

The `va_start` macro initializes a variable argument list so that additional arguments to the variadic function after the fixed parameters can be captured in the list. According to the C11 and C++14 Standards, if you use one of the flagged data types for the second argument of the `va_start` macro (for instance, `rightmostFixedArg` in the preceding example), the behavior is undefined.

If the data type involves a nontrivial copy constructor, the behavior is implementation-defined. For instance, whether the copy constructor is invoked in the call to `va_start` depends on the compiler.

Fix

When using the `va_start` macro, try to use the types `int`, `unsigned int` or `double` for the rightmost named parameter of the variadic function. Then, use this parameter as the second argument of the `va_start` macro.

For instance, in this example, the rightmost named parameter of the variadic function has a supported data type `int`:

```
void multipleArgumentFunction(int someArg, int rightmostFixedArg, ...) {
    va_list myList;
    va_start(myList, rightmostFixedArg);
    ...
    va_end(myList);
}
```

To avoid undefined and implementation-defined behavior, minimize the use of variadic functions. Use the checkers for `MISRA C:2012 Rule 17.1` or `MISRA C++:2008 Rule 8-4-1` to detect use of variadic functions.

Examples

Incorrect Data Types for Second Argument of `va_start`

```
#include <string>
#include <cstdarg>

double addVariableNumberOfDoubles(double* weight, short num, ...) {
    double sum=0.0;
    va_list list;
    va_start(list, num);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, double);
    }
    va_end(list);
    return sum;
}

double addVariableNumberOfFloats(float* weight, int num, std::string s, ...) {
    float sum=0.0;
    va_list list;
    va_start(list, s);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, float);
    }
    va_end(list);
    return sum;
}
```

In this example, the checker flags the call to `va_start` in:

- `addVariableNumberOfDoubles` because the argument has type `short`, which undergoes default argument promotion to `int`.
- `addVariableNumberOfFloats` because the argument has type `std::string`, which has a nontrivial copy constructor.

Correction — Fix Data Type for Second Argument of `va_start`

Make sure that the second argument of the `va_start` macro has a supported data type. In the following corrected example:

- In `addVariableNumberOfDoubles`, the data type of the last named parameter of the variadic function is changed to `int`.
- In `addVariableNumberOfFloats`, the second and third parameters of the variadic function are switched so that data type of the last named parameter is `int`.

```
#include <string>
#include <cstdarg>

double addVariableNumberOfDoubles(double* weight, int num, ...) {
    double sum=0.0;
    va_list list;
```

```
    va_start(list, num);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, double);
    }
    va_end(list);
    return sum;
}

double addVariableNumberOfFloats(double* weight, std::string s, int num, ...) {
    double sum=0.0;
    va_list list;
    va_start(list, num);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, double);
    }
    va_end(list);
    return sum;
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: VA_START_INCORRECT_TYPE
Impact: Medium

See Also

Find defects (-checkers)|Incorrect data type passed to va_arg|Incorrect use of va_start|Too many va_arg calls for current argument list

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019a

Incorrect use of offsetof in C++

Incorrect arguments to `offsetof` macro causes undefined behavior

Description

This defect occurs when you pass arguments to the `offsetof` macro for which the behavior of the macro is not defined.

The `offsetof` macro:

```
offsetof(classType, aMember)
```

returns the offset in bytes of the data member `aMember` from the beginning of an object of type `classType`. For use in `offsetof`, `classType` and `aMember` have certain restrictions:

- `classType` must be a standard layout class.

  For instance, it must not have `virtual` member functions. For more information on the requirements for a standard layout class, see C++ named requirements: StandardLayoutType.
- `aMember` must not be static.
- `aMember` must not be a member function.

The checker flags uses of the `offsetof` macro where the arguments violate one or more of these restrictions.

Risk

Violating the restrictions on the arguments of the `offsetof` macro leads to undefined behavior.

Fix

Use the `offsetof` macro only on nonstatic data members of a standard layout class.

The result details state which restriction on the `offsetof` macro is violated. Fix the violation.

Examples

Use of `offsetof` Macro with Nonstandard Layout Class

```
#include <cstddef>

class myClass {
     int member1;
  public:
     int member2;
};

void func() {
  size_t off = offsetof(myClass, member2);
  // ...
}
```

In this example, the class `myClass` has two data members with different access control, one private and the other public. Therefore, the class does not satisfy the requirements of a standard layout class and cannot be used with the `offsetof` macro.

Correction — Use Uniform Access Control for All Data Members

If the use of `offsetof` is important for the application, make sure that the first argument is a class with a standard layout. For instance, see if you can work around the need for a public data member.

```
#include <cstddef>

class myClass {
     int member1;
     int member2;
  public:
     int getMember2(void) { return member2;}
     friend void func(void);
};

void func() {
  size_t off = offsetof(myClass, member2);
  // ...
}
```

Result Information

Group: Programming
Language: C++
Default: On
Command-Line Syntax: `OFFSETOF_MISUSE`
Impact: Medium

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019a

Incorrect use of va_start

`va_start` is called in a non-variadic function or called with a second argument that is not the rightmost parameter of a variadic function

Description

This defect occurs when you use the `va_start` macro in a way that violates its specifications.

In a variadic function or function with variable number of arguments:

```
void multipleArgumentFunction(int someArg, int rightmostFixedArg, ...) {
    va_list myList;
    va_start(myList, rightmostFixedArg);
    ...
    va_end(myList);
}
```

The `va_start` macro initializes a variable argument list so that additional arguments to the variadic function after the fixed parameters can be captured in the list. In the preceding example, the `va_start` macro initializes `myList` so that it can capture arguments after `rightmostFixedArg`.

You can violate the specifications of `va_start` in multiple ways. For instance:

- You call `va_start` in a non-variadic function.
- The second argument of `va_start` is not the rightmost fixed parameter of the variadic function.

Risk

Violating the specifications of the `va_start` macro can result in compilation errors. If the compiler fails to detect the violation, the violation can result in undefined behavior.

Fix

Make sure that:

- The `va_start` macro is used in a variadic function
- The second argument of the `va_start` macro is the rightmost fixed parameter of the variadic function.

To avoid undefined and implementation-defined behavior, minimize the use of variadic functions. Use the checkers for `MISRA C:2012 Rule 17.1` or `MISRA C++:2008 Rule 8-4-1` to detect use of variadic functions.

Examples

Incorrect Argument to `va_start`

```
#include <stdarg.h>

double addVariableNumberOfDoubles(int num, double* weight, ...) {
    double sum=0.0;
    va_list list;
```

```
    va_start(list, num);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, double);
    }
    va_end(list);
    return sum;
}
```

In this example, the rightmost fixed parameter to the `addVariableNumberOfDoubles` function is `weight`. However, a different parameter is used as the second argument to the `va_start` macro.

Correction — Switch Order of Fixed Parameters of Variadic Function

One possible correction is to modify the order of fixed parameters to the variadic function so that the rightmost fixed parameter is used for the `va_start` macro.

```
#include <stdarg.h>

double addVariableNumberOfDoubles(double* weight, int num, ...) {
    double sum=0.0;
    va_list list;
    va_start(list, num);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, double);
    }
    va_end(list);
    return sum;
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `VA_START_MISUSE`
Impact: Medium

See Also

`Find defects (-checkers)` | `Incorrect data type passed to va_arg` | `Incorrect type data passed to va_start` | `Too many va_arg calls for current argument list`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019a

Inline constraint not respected

Modifiable static variable is modified in nonstatic inline function

Description

This defect occurs when you refer to a file scope modifiable static variable or define a local modifiable static variable in a nonstatic inlined function. The checker considers a variable as modifiable if it is not `const`-qualified.

For instance, `var` is a modifiable `static` variable defined in an `inline` function `func`. `g_step` is a file scope modifiable static variable referred to in the same inlined function.

```
static int g_step;
inline void func (void) {
   static int var = 0;
   var += g_step;
}
```

Risk

When you modify a static variable in multiple function calls, you expect to modify the same variable in each call. For instance, each time you call `func`, the same instance of `var1` is incremented but a separate instance of `var2` is incremented.

```
void func(void) {
   static var1 = 0;
   var2 = 0;
   var1++;
   var2++;
}
```

If a function has an inlined and non-inlined definition (in separate files), when you call the function, the C standard allows compilers to use either the inlined or the non-inlined form (see ISO/IEC 9899:2011, sec. 6.7.4). If your compiler uses an inlined definition in one call and the non-inlined definition in another, you are no longer modifying the same variable in both calls. This behavior defies the expectations from a static variable.

Fix

Use one of these fixes:

- If you do not intend to modify the variable, declare it as `const`.

  If you do not modify the variable, there is no question of unexpected modification.
- Make the variable non-`static`. Remove the `static` qualifier from the declaration.

  If the variable is defined in the function, it becomes a regular local variable. If defined at file scope, it becomes an extern variable. Make sure that this change in behavior is what you intend.
- Make the function `static`. Add a `static` qualifier to the function definition.

If you make the function `static`, the file with the inlined definition always uses the inlined definition when the function is called. Other files use another definition of the function. The question of which function definition gets used is not left to the compiler.

Examples

Static Variable Use in Inlined and External Definition

```
/* file1. c  : contains inline definition of get_random()*/

inline unsigned int get_random(void)
{

    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}


int call_get_random(void)
{
    unsigned int rand_no;
    int ii;
    for (ii = 0; ii < 100; ii++) {
         rand_no = get_random();
    }
    rand_no = get_random();
    return 0;
}

/* file2. c  : contains external definition of get_random()*/

extern unsigned int get_random(void)
{
    /* Initialize seeds */
    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}
```

In this example, `get_random()` has an inline definition in `file1.c` and an external definition in `file2.c`. When `get_random` is called in `file1.c`, compilers are free to choose whether to use the inline or the external definition.

Depending on the definition used, you might or might not modify the version of `m_z` and `m_w` in the inlined version of `get_random()`. This behavior contradicts the usual expectations from a static variable. When you call `get_random()`, you expect to always modify the same `m_z` and `m_w`.

Correction — Make Inlined Function Static

One possible correction is to make the inlined `get_random()` static. Irrespective of your compiler, calls to `get_random()` in `file1.c` then use the inlined definition. Calls to `get_random()` in other files use the external definition. This fix removes the ambiguity about which definition is used and whether the static variables in that definition are modified.

```
/* file1. c  : contains inline definition of get_random()*/

static inline unsigned int get_random(void)
{

    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}


int call_get_random(void)
{
    unsigned int rand_no;
    int ii;
    for (ii = 0; ii < 100; ii++) {
         rand_no = get_random();
    }
    rand_no = get_random();
    return 0;
}

/* file2. c  : contains external definition of get_random()*/

extern unsigned int get_random(void)
{
    /* Initialize seeds */
    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `INLINE_CONSTRAINT_NOT_RESPECTED`
Impact: Medium

See Also

`Find defects (-checkers)`

Topics
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Invalid assumptions about memory organization

Address is computed by adding or subtracting from address of a variable

Description

This defect occurs when you compute the address of a variable in the stack by adding or subtracting from the address of another non-array variable.

Risk

When you compute the address of a variable in the stack by adding or subtracting from the address of another variable, you assume a certain memory organization. If your assumption is incorrect, accessing the computed address can be invalid.

Fix

Do not perform an access that relies on assumptions about memory organization.

Examples

Reliance on Memory Organization

```
void func(void) {
    int var1 = 0x00000011, var2;
    *(&var1 + 1) = 0;
}
```

In this example, the programmer relies on the assumption that `&var1 + 1` provides the address of `var2`. Therefore, an **Invalid assumptions about memory organization** appears on the `+` operation. In addition, a **Pointer access out of bounds** error also appears on the dereference.

Correction — Do Not Rely on Memory Organization

One possible correction is not perform direct computation on addresses to access separately declared variables.

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `INVALID_MEMORY_ASSUMPTION`
Impact: Medium
CWE ID: 188

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Invalid file position

`fsetpos()` is invoked with a file position argument not obtained from `fgetpos()`

Description

This defect occurs when the file position argument of `fsetpos()` uses a value that is not obtained from `fgetpos()`.

Risk

The function `fgetpos(FILE *stream, fpos_t *pos)` gets the current file position of the stream. When you use any other value as the file position argument of `fsetpos(FILE *stream, const fpos_t *pos)`, you might access an unintended location in the stream.

Fix

Use the value returned from a successful call to `fgetpos()` as the file position argument of `fsetpos()`.

Examples

`memset()` Sets File Position Argument

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


FILE *func(FILE *file)
{
    fpos_t offset;
    if (file == NULL)
    {
        /* Handle error */
    }
    /* Store initial position in variable 'offset' */
    (void)memset(&offset, 0, sizeof(offset));

    /* Read data from file */

    /* Return to the initial position. offset was not
    returned from a call to fgetpos()    */
    if (fsetpos(file, &offset) != 0)
    {
        /* Handle error */
    }
    return file;
}
```

In this example, `fsetpos()` uses `offset` as its file position argument. However, the value of `offset` is set by `memset()`. The preceding code might access the wrong location in the stream.

Correction — Use a File Position Returned From `fgetpos()`

Call `fgetpos()`, and if it returns successfully, use the position argument in your call to `fsetpos()`.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

FILE *func(FILE *file)
{
    fpos_t offset;
    if (file == NULL)
    {
        /* Handle error */
    }
    /* Store initial position in variable 'offset'
    using fgetpos() */
    if (fgetpos(file, &offset) != 0)
    {
        /* Handle error */
    }

    /* Read data from file */

    /* Back to the initial position */
    if (fsetpos(file, &offset) != 0)
    {
        /* Handle error */
    }
    return file;
}
```

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: `INVALID_FILE_POS`
Impact: Medium

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Invalid use of = operator

Assignment in conditional statement

Description

This defect occurs when an assignment is made inside the predicate of a conditional, such as `if` or `while`.

In C and C++, a single equal sign is an assignment not a comparison. Using a single equal sign in a conditional statement can indicate a typo or a mistake.

Risk

- Conditional statement tests the wrong values— The single equal sign operation assigns the value of the right operand to the left operand. Then, because this assignment is inside the predicate of a conditional, the program checks whether the new value of the left operand is nonzero or not NULL.
- Maintenance and readability issues — Even if the assignment is intended, someone reading or updating the code can misinterpret the assignment as an equality comparison instead of an assignment.

Fix

- If the assignment is a bug, to check for equality, add a second equal sign (==).
- If the assignment inside the conditional statement was intentional, to improve readability, separate the assignment and the test. Move the assignment outside the control statement. In the control statement, simply test the result of the assignment.

  If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Single Equal Sign Inside an `if` Condition

```
#include <stdio.h>

void bad_equals_ex(int alpha, int beta)
{
    if(alpha = beta)
    {
        printf("Equal\n");
    }
}
```

The equal sign is flagged as a defect because the assignment operator is used within the predicate of the if-statement. The predicate assigns the value `beta` to `alpha`, then implicitly tests whether `alpha` is true or false.

Correction — Change Expression to Comparison

One possible correction is adding an additional equal sign. This correction changes the assignment to a comparison. The if condition compares whether `alpha` and `beta` are equal.

```
#include <stdio.h>

void equality_test(int alpha, int beta)
{
    if(alpha == beta)
    {
        printf("Equal\n");
    }
}
```

Correction — Assignment and Comparison Inside the `if` Condition

If an assignment must be made inside the predicate, a possible correction is adding an explicit comparison. This correction assigns the value of `beta` to `alpha`, then explicitly checks whether `alpha` is nonzero. The code is clearer.

```
#include <stdio.h>

int assignment_not_zero(int alpha, int beta)
{
    if((alpha = beta) != 0)
    {
        return alpha;
    }
    else
    {
        return 0;
    }
}
```

Correction — Move Assignment Outside the `if` Statement

If the assignment can be made outside the control statement, one possible correction is to separate the assignment and comparison. This correction assigns the value of `beta` to `alpha` before the if. Inside the if-condition, only `alpha` is given to test if `alpha` is nonzero or not NULL.

```
#include <stdio.h>

void assign_and_print(int alpha, int beta)
{
    alpha = beta;
    if(alpha)
    {
        printf("%d", alpha);
    }
}
```

Check Information

Group: Programming
Language: C | C++
Default: On for handwritten code, off for generated code

Command-Line Syntax: BAD_EQUAL_USE
Impact: Medium
CWE ID: 480, 481

See Also

Find defects (-checkers) | Invalid use of == (equality) operator

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Invalid use of == operator

Equality operation in assignment statement

Description

This defect occurs when you use an equality operator instead of an assignment operator in a simple statement.

Risk

The use of == operator instead of an = operator can silently produce incorrect results. If you intended to assign a value to a variable, the assignment does not occur. The variable retains its previous value or if not initialized previously, stays uninitialized.

Fix

Use the = (assignment) operator instead of the == (equality) operator.

The check appears on chained assignment and equality operators such as:

```
compFlag = val1 == val2;
```

For better readability of your code, place the equality check in parenthesis.

```
compFlag = (val1 == val2);
```

If the use of == operator is intended, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Equality Evaluation in `for`-Loop

```
void populate_array(void)
{
    int i = 0;
    int j = 0;
    int array[4];

    for (j == 5; j < 9; j++) {
        array[i] = j;
        i++;
    }
}
```

Inside the `for`-loop, the statement `j == 5` tests whether `j` is equal to 5 instead of setting `j` to 5. The `for`-loop iterates from 0 to 8 because `j` starts with a value of 0, not 5. A by-product of the invalid equality operator is an out-of-bounds array access in the next line.

Correction — Change to Assignment Operator

One possible correction is to change the == operator to a single equal sign (=). Changing the == sign resolves both defects because the `for`-loop iterates the intended number of times.

```
void populate_array(void)
{
    int i = 0;
    int j = 0;
    int array[4];

    for (j = 5; j < 9; j++) {
        array[i] = j;
        i++;
    }
}
```

Check Information

Group: Programming
Language: C | C++
Default: On for handwritten code, off for generated code
Command-Line Syntax: BAD_EQUAL_EQUAL_USE
Impact: High
CWE ID: 480, 482

See Also

`Find defects (-checkers)` | `Invalid use of = (assignment) operator`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Invalid use of standard library routine

Wrong arguments to standard library function

Description

This defect occurs when you use invalid arguments with a function from the standard library. This defect picks up errors related to other functions not covered by float, integer, memory, or string standard library routines.

Risk

Invalid arguments to a standard library function result in undefined behavior.

Fix

The fix depends on the root cause of the defect. For instance, the argument to a `printf` function can be `NULL` because a pointer was initialized with `NULL` and the initialization value was not overwritten along a specific execution path.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Calling `printf` Without a String

```
#include <stdio.h>
#include <stdlib.h>

void print_null(void) {

  printf(NULL);
}
```

The function `printf` takes only string input arguments or format specifiers. In this function, the input value is NULL, which is not a valid string.

Correction — Use Compatible Input Arguments

One possible correction is to change the input arguments to fit the requirements of the standard library routine. In this example, the input argument was changed to a character.

```
#include <stdio.h>

void print_null(void) {
    char zero_val = '0';
    printf((const char*)zero_val);
}
```

Check Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `OTHER_STD_LIB`
Impact: High
CWE ID: 227, 690

See Also

`Find defects (-checkers)` | `Invalid use of standard library floating point routine` | `Invalid use of standard library integer routine` | `Invalid use of standard library memory routine` | `Invalid use of standard library string routine`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Invalid va_list argument

Variable argument list used after invalidation with `va_end` or not initialized with `va_start` or `va_copy`

Description

This defect occurs when you use a `va_list` variable as an argument to a function in the `vprintf` group but:

- You do not initialize the variable previously using `va_start` or `va_copy`.
- You invalidate the variable previously using `va_end` and do not reinitialize it.

For instance, you call the function `vsprintf` as `vsprintf (buffer,format, args)`. However, before the function call, you do not initialize the `va_list` variable `args` using either of the following:

- `va_start(args, paramName)`. `paramName` is the last named argument of a variable-argument function. For instance, for the function definition `void func(int n, char c, ...) {}`, `c` is the last named argument.
- `va_copy(args, anotherList)`. `anotherList` is another valid `va_list` variable.

Risk

The behavior of an uninitialized `va_list` argument is undefined. Calling a function with an uninitialized `va_list` argument can cause stack overflows.

Fix

Before using a `va_list` variable as function argument, initialize it with `va_start` or `va_copy`.

Clean up the variable using `va_end` only after all uses of the variable.

Examples

`va_list` Variable Used Following Call to `va_end`

```
#include <stdarg.h>
#include <stdio.h>

int call_vfprintf(int line, const char *format, ...) {
    va_list ap;
    int r=0;

    va_start(ap, format);
    r = vfprintf(stderr, format, ap);
    va_end(ap);

    r += vfprintf(stderr, format, ap);
    return r;
}
```

In this example, the `va_list` variable `ap` is used in the `vfprintf` function, after the `va_end` macro is called.

Correction — Call va_end After Using va_list Variable

One possible correction is to call `va_end` only after all uses of the `va_list` variable.

```
#include <stdarg.h>
#include <stdio.h>

int call_vfprintf(int line, const char *format, ...) {
    va_list ap;
    int r=0;

    va_start(ap, format);
    r = vfprintf(stderr, format, ap);
    r += vfprintf(stderr, format, ap);
    va_end(ap);

    return r;
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `INVALID_VA_LIST_ARG`
Impact: High
CWE ID: 628

See Also

`Find defects (-checkers)` | `Incorrect data type passed to va_arg`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Memory comparison of float-point values

Object representation of floating-point values can be different (same) for equal (not equal) floating-point values

Description

This defect occurs when you compare the object representation of floating-point values or the object representation of structures containing floating-point members. When you use the functions `memcmp`, `bcmp`, or `wmemcmp` to perform the bit pattern comparison, the defect is raised.

Risk

The object representation of floating-point values uses specific bit patterns to encode those values. Floating-point values that are equal, for instance `-0.0` and `0.0` in the IEC 60559 standard, can have different bit patterns in their object representation. Similarly, floating-point values that are not equal can have the same bit pattern in their object representation.

Fix

When you compare structures containing floating-point members, compare the structure members individually.

To compare two floating-point values, use the == or != operators. If you follow a standard that discourages the use of these operators, such as MISRA, ensure that the difference between the floating-point values is within an acceptable range.

Examples

Using `memcmp` to Compare Structures with Floating-Point Members

```
#include <string.h>

typedef struct {
    int i;
    float f;
} myStruct;

extern void initialize_Struct(myStruct *);

int func_cmp(myStruct *s1, myStruct *s2) {
/* Comparison between structures containing
* floating-point members */
    return memcmp
        ((const void *)s1, (const void *)s2, sizeof(myStruct));
}

void func(void) {
    myStruct s1, s2;
    initialize_Struct(&s1);
    initialize_Struct(&s2);
    (void)func_cmp(&s1, &s2);
}
```

In this example, `func_cmp()` calls `memcmp()` to compare the object representations of structures `s1` and `s2`. The comparison might be inaccurate because the structures contain floating-point members.

Correction — Compare Structure Members Individually

One possible correction is to compare the structure members individually and to ensure that the difference between the floating-point values is within an acceptable range defined by `ESP`.

```
#include <string.h>
#include <math.h>

typedef struct {
    int i;
    float f;
} myStruct;

extern void initialize_Struct(myStruct *);

#define ESP 0.00001

int func_cmp(myStruct *s1, myStruct *s2) {

/*Structure members are compared individually */
    return ((s1->i == s2->i) &&
            (fabsf(s1->f - s2->f) <= ESP));
}

void func(void) {
    myStruct s1, s2;
    initialize_Struct(&s1);
    initialize_Struct(&s2);
    (void)func_cmp(&s1, &s2);
}
```

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: `MEMCMP_FLOAT`
Impact: Low

See Also

`Find defects (-checkers)` | `Floating point comparison with equality operators` | `Memory comparison of padding data` | `Memory comparison of strings`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Memory comparison of padding data

`memcmp` compares data stored in structure padding

Description

This defect occurs when you use the `memcmp` function to compare two structures as a whole. In the process, you compare meaningless data stored in the structure padding.

For instance:

```
struct structType {
    char member1;
    int member2;
    .
    .
};

structType var1;
structType var2;
.
.
if(memcmp(&var1,&var2,sizeof(var1)))
{...}
```

Risk

If members of a structure have different data types, your compiler introduces additional padding for data alignment in memory. For an example of padding, see `Higher Estimate of Local Variable Size`.

The content of these extra padding bytes is meaningless. The C Standard allows the content of these bytes to be indeterminate, giving different compilers latitude to implement their own padding. If you perform a byte-by-byte comparison of structures with `memcmp`, you compare even the meaningless data stored in the padding. You might reach the false conclusion that two data structures are not equal, even if their corresponding members have the same value.

Fix

Instead of comparing two structures in one attempt, compare the structures member by member.

For efficient code, write a function that does the comparison member by member. Use this function for comparing two structures.

You can use `memcmp` for byte-by-byte comparison of structures only if you know that the structures do not contain padding. Typically, to prevent padding, you use specific attributes or pragmas such as `#pragma pack`. However, these attributes or pragmas are not supported by all compilers and make your code implementation-dependent. If your structures contain bit-fields, using these attributes or pragmas cannot prevent padding.

Examples

Structures Compared with `memcmp`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{

    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    if (0 == memcmp(left, right, sizeof(S_Padding)))
    {
        return 1;
    }
    else
        return 0;
}
```

In this example, `memcmp` compares byte-by-byte the two structures that `left` and `right` point to. Even if the values stored in the structure members are the same, the comparison can show an inequality if the meaningless values in the padding bytes are not the same.

Correction — Compare Structures Member by Member

One possible correction is to compare individual structure members.

Note You can compare entire arrays by using `memcmp`. All members of an array have the same data type. Padding bytes are not required to store arrays.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
```

```
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{
    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    return ((left->c == right->c) &&
            (left->i == right->i) &&
            (left->bf1 == right->bf1) &&
            (left->bf2 == right->bf2) &&
            (memcmp(left->buffer, right->buffer, 20) == 0));
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `MEMCMP_PADDING_DATA`
Impact: Medium
CWE ID: 188

See Also

`Find defects (-checkers)|Memory comparison of strings`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Memory comparison of strings

`memcmp` compares data stored in strings after the null terminator

Description

This defect occurs when:

- You compare two strings byte-by-byte with the `memcmp` function.
- The number of bytes compared is such that you compare meaningless data stored after the null terminator.

For instance:

```
memcmp(string1, string2, sizeof(string1))
```

can compare bytes in the string after the null terminator.

Risk

The null terminator signifies the end of a string. Comparison of bytes after the null terminator is meaningless. You might reach the false conclusion that two strings are not equal, even if the bytes before the null terminator store the same value.

Fix

Use `strcmp` for string comparison. The function compares strings only up to the null terminator.

If you use `memcmp` for a byte-by-byte comparison of two strings, avoid comparison of bytes after the null terminator. Determine the number of bytes to compare by using the `strlen` function.

Examples

Strings Compared with `memcmp`

```
#include <stdio.h>
#include <string.h>

#define SIZE20 20

int func()
{
    char s1[SIZE20] =  "abc";
    char s2[SIZE20] =  "abc";

    return memcmp(s1, s2, sizeof(s1));
}
```

In this example, `sizeof` returns the length of the entire array `s1`, which is 20. However, only the first three bytes of the string are relevant.

Even though `s1` and `s2` hold the same value, the comparison with `memcmp` can show a false inequality.

Correction — Use `strlen` to Determine Number of Bytes to Compare

One possible correction is to determine the number of bytes to compare using the `strlen` function. `strlen` returns the number of bytes *before* the null terminator (and excluding the null terminator itself).

```
#include <stdio.h>
#include <string.h>

#define SIZE20 20

int func()
{
    char s1[SIZE20] =  "abc";
    char s2[SIZE20] =  "abc";

    return memcmp(s1, s2, strlen(s1));
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `MEMCMP_STRINGS`
Impact: Medium
CWE ID: 188

See Also

`Find defects (-checkers)` | `Memory comparison of padding data`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Missing byte reordering when transferring data

Different endianness of host and network

Description

This defect occurs when you do not use a byte ordering function:

- Before sending data to a network socket.
- After receiving data from a network socket.

Risk

Some system architectures implement little endian byte ordering (least significant byte first), and other systems implement big endian (most significant byte first). If the endianness of the sent data does not match the endianness of the receiving system, the value returned when reading the data is incorrect.

Fix

After receiving data from a socket, use a byte ordering function such as `ntohl()`. Before sending data to a socket, use a byte ordering function such as `htonl()` .

Examples

Data Transferred Without Byte Reordering

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <byteswap.h>
#include <unistd.h>
#include <string.h>


unsigned int func(int sock, int server)
{
    unsigned int num;   /* assume int is 32-bits */
    if (server)
    {
        /* Server side */
        num = 0x17;
        /* Endianness of server host may not match endianness of network. */
        if (send(sock, (void *)&num, sizeof(num), 0) < (int)sizeof(num))
        {
            /* Handle error */
        }
        return 0;
    }
    else {
```

```
        /* Endianness of client host may not match endianness of network. */
        if (recv (sock, (void *)&num, sizeof(num), 0) < (int) sizeof(num))
        {
            /* Handle error */
        }

        /* Comparison may be inaccurate */
        if (num> 255)
        {
            return 255;
        }
        else
        {
            return num;
        }
    }
}
```

In this example, variable `num` is assigned hexadecimal value `0x17` and is sent over a network to the client from the server. If the server host is little endian and the network is big endian, `num` is transferred as `0x17000000`. The client then reads an incorrect value for `num` and compares it to a local numeric value.

Correction — Use Byte Ordering Function

Before sending `num` from the server host, use `htonl()` to convert from host to network byte ordering. Similarly, before reading `num` on the client host, use `ntohl()` to convert from network to host byte ordering.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <byteswap.h>
#include <unistd.h>
#include <string.h>

unsigned int func(int sock, int server)
{
    unsigned int num;   /* assume int is 32-bits */
    if (server)
    {
        /* Server side */
        num = 0x17;

        /* Convert to network byte order. */
        num = htonl(num);
        if (send(sock, (void *)&num, sizeof(num), 0) < (int)sizeof(num))
        {
            /* Handle error */
        }
        return 0;
    }
    else {
```

```
        if (recv (sock, (void *)&num, sizeof(num), 0) < (int) sizeof(num))
        {
            /* Handle error */
        }

        /* Convert to host byte order. */
        num = ntohl(num);
        if (num > 255)
        {
            return 255;
        }
        else
        {
            return num;
        }
    }
}
```

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: `MISSING_BYTESWAP`
Impact: Medium
CWE ID: 188, 198

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Missing null in string array

String does not terminate with null character

Description

This defect occurs when a string does not have enough space to terminate with a null character `'\0'`.

This defect applies only for projects in C.

Risk

A buffer overflow can occur if you copy a string to an array without assuming the implicit null terminator.

Fix

If you initialize a character array with a literal, avoid specifying the array bounds.

```
char three[]  = "THREE";
```

The compiler automatically allocates space for a null terminator. In the preceding example, the compiler allocates sufficient space for five characters and a null terminator.

If the issue occurs after initialization, you might have to increase the size of the array by one to account for the null terminator.

In certain circumstances, you might want to initialize the character array with a sequence of characters instead of a string. In this situation, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Array size is too small

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[5] = "THREE";
}
```

The character array `three` has a size of 5 and 5 characters `'T'`, `'H'`, `'R'`, `'E'`, and `'E'`. There is no room for the null character at the end because `three` is only five bytes large.

Correction — Increase Array Size

One possible correction is to change the array size to allow for the five characters plus a null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
```

```
    static char two[5]   = "TWO";
    static char three[6] = "THREE";
}
```

Correction — Change Initialization Method

One possible correction is to initialize the string by leaving the array size blank. This initialization method allocates enough memory for the five characters and a terminating-null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[]  = "THREE";
}
```

Check Information

Group: Programming
Language: C
Default: On for handwritten code, off for generated code
Command-Line Syntax: `MISSING_NULL_CHAR`
Impact: Low
CWE ID: 170

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Misuse of a FILE object

Use of copy of FILE object

Description

This defect occurs when:

- You dereference a pointer to a FILE object, including indirect dereference by using `memcmp()`.
- You modify an entire FILE object or one of its components through its pointer.
- You take the address of FILE object that was not returned from a call to an `fopen`-family function. No defect is raised if a macro defines the pointer as the address of a built-in FILE object, such as `#define ptr (&__stdout)`.

Risk

In some implementations, the address of the pointer to a FILE object used to control a stream is significant. A pointer to a copy of a FILE object is interpreted differently than a pointer to the original object, and can potentially result in operations on the wrong stream. Therefore, the use of a copy of a FILE object can cause the software to stop responding, which an attacker might exploit in denial-of-service attacks.

Fix

Do not make a copy of a FILE object. Do not use the address of a FILE object that was not returned from a successful call to an `fopen`-family function.

Examples

Copy of FILE Object Used in `fputs()`

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

void fatal_error(void);

int func(void)
{
    /*'stdout' dereferenced and contents
        copied to 'my_stdout'. */
    FILE my_stdout = *stdout;

    /* Address of 'my_stdout' may not point to correct stream. */
    if (fputs("Hello, World!\n", &my_stdout) == EOF)
    {
        /* Handler error */
        fatal_error();
    }
    return 0;
```

```
}
```

In this example, FILE object `stdout` is dereferenced and its contents are copied to `my_stdout`. The contents of `stdout` might not be significant. `fputs()` is then called with the address of `my_stdout` as an argument. Because no call to `fopen()` or a similar function was made, the address of `my_stdout` might not point to the correct stream.

Correction — Copy the FILE Object Pointer

Declare `my_stdout` to point to the same address as `stdout` to ensure that you write to the correct stream when you call `fputs()`.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

void fatal_error(void);

int func(void)
{
    /* 'my_stdout' and 'stdout' point to the same object. */
    FILE *my_stdout = stdout;
    if (fputs("Hello, World!\n", my_stdout) == EOF)
    {
        /* Handler error */
        fatal_error();
    }
    return 0;
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `FILE_OBJECT_MISUSE`
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Misuse of errno

`errno` incorrectly checked for error conditions

Description

This defect occurs when you check `errno` for error conditions in situations where checking `errno` does not guarantee the absence of errors. In some cases, checking `errno` can lead to false positives.

For instance, you check `errno` following calls to the functions:

- `fopen`: If you follow the ISO Standard, the function might not set `errno` on errors.
- `atof`: If you follow the ISO Standard, the function does not set `errno`.
- `signal`: The `errno` value indicates an error only if the function returns the `SIG_ERR` error indicator.

Risk

The ISO C Standard does not enforce that these functions set `errno` on errors. Whether the functions set `errno` or not is implementation-dependent.

To detect errors, if you check `errno` alone, the validity of this check also becomes implementation-dependent.

In some cases, the `errno` value indicates an error only if the function returns a specific error indicator. If you check `errno` before checking the function return value, you can see false positives.

Fix

For information on how to detect errors, see the documentation for that specific function.

Typically, the functions return an out-of-band error indicator to indicate errors. For instance:

- `fopen` returns a null pointer if an error occurs.
- `signal` returns the `SIG_ERR` error indicator and sets `errno` to a positive value. Check `errno` only after you have checked the function return value.

Examples

Incorrectly Checking for `errno` After `fopen` Call

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define fatal_error() abort()

const char *temp_filename = "/tmp/demo.txt";

FILE *func()
{
    FILE *fileptr;
```

```
    errno = 0;
    fileptr = fopen(temp_filename, "w+b");
    if (errno != 0) {
        if (fileptr != NULL) {
            (void)fclose(fileptr);
        }
        /* Handle error */
        fatal_error();
    }
    return fileptr;
}
```

In this example, `errno` is the first variable that is checked after a call to `fopen`. You might expect that `fopen` changes `errno` to a nonzero value if an error occurs. If you run this code with an implementation of `fopen` that does not set `errno` on errors, you might miss an error condition. In this situation, `fopen` can return a null pointer that escapes detection.

Correction — Check Return Value of `fopen` After Call

One possible correction is to only check the return value of `fopen` for a null pointer.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define fatal_error() abort()

const char *temp_filename = "/tmp/demo.txt";

FILE *func()
{
    FILE *fileptr;
    fileptr = fopen(temp_filename, "w+b");
    if (fileptr == NULL) {
        fatal_error();
    }
    return fileptr;
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `ERRNO_MISUSE`
Impact: High
CWE ID: 703

See Also

`Errno not checked` | `Errno not reset` | `Find defects (-checkers)` | `Returned value of a sensitive function not checked` | `Unsafe conversion from string to numerical value`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Misuse of errno in a signal handler

You read `errno` after calling an `errno`-setting function in a signal handler

Description

This defect occurs when you call one of these functions in a signal handler:

- `signal`: You call the `signal` function in a signal handler and then read the value of `errno`.

  For instance, the signal handler function `handler` calls `signal` and then calls `perror`, which reads `errno`.

  ```
  void handler(int signum) {
    pfv old_handler = signal(signum, SIG_DFL);
    if (old_handler == SIG_ERR) {
      perror("SIGINT handler");
    }
  }
  ```

- `errno`-setting POSIX function: You call an `errno`-setting POSIX function in a signal handler but do not restore `errno` when returning from the signal handler.

  For instance, the signal handler function `handler` calls `waitpid`, which changes `errno`, but does not restore `errno` before returning.

  ```
  void handler(int signum) {
    int rc = waitpid(-1, NULL, WNOHANG);
    if (ECHILD != errno) {
    }
  }
  ```

Risk

In each case that the checker flags, you risk relying on an indeterminate value of `errno`.

- `signal`: If the call to `signal` in a signal handler fails, the value of `errno` is indeterminate (see C11 Standard, Sec. 7.14.1.1). If you rely on a specific value of `errno`, you can see unexpected results.
- `errno`-setting POSIX function: An `errno`-setting function sets `errno` on failure. If you read `errno` after a signal handler is called and the signal handler itself calls an `errno`-setting function, you can see unexpected results.

Fix

Avoid situations where you risk relying on an indeterminate value of `errno`.

- `signal`: After calling the `signal` function in a signal handler, do not read `errno` or use a function that reads `errno`.
- `errno`-setting POSIX function: Before calling an `errno`-setting function in a signal handler, save `errno` to a temporary variable. Restore `errno` from this variable before returning from the signal handler.

Examples

Reading `errno` After `signal` Call in Signal Handler

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

#define fatal_error() abort()

void handler(int signum) {
    if (signal(signum, SIG_DFL) == SIG_ERR) {
        perror("SIGINT handler");
    }
}

int func(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
        fatal_error();
    }
    /* Program code */
    if (raise(SIGINT) != 0) {
        /* Handle error */
        fatal_error();
    }
    return 0;
}
```

In this example, the function `handler` is called to handle the `SIGINT` signal. In the body of `handler`, the `signal` function is called. Following this call, the value of `errno` is indeterminate. The checker raises a defect when the `perror` function is called because `perror` relies on the value of `errno`.

Correction — Avoid Reading `errno` After `signal` Call

One possible correction is to not read `errno` after calling the `signal` function in a signal handler. The corrected code here calls the `abort` function via the `fatal_error` macro instead of the `perror` function.

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

#define fatal_error() abort()

void handler(int signum) {
    if (signal(signum, SIG_DFL) == SIG_ERR) {
        fatal_error();
    }
}

int func(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
        fatal_error();
    }
    /* Program code */
    if (raise(SIGINT) != 0) {
```

```
        /* Handle error */
        fatal_error();
    }
    return 0;
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `SIG_HANDLER_ERRNO_MISUSE`
Impact: Medium

See Also

`Errno not checked` | `Errno not reset` | `Find defects (-checkers)` | `Function called from signal handler not asynchronous-safe`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Misuse of narrow or wide character string

Narrow (wide) character string passed to wide (narrow) string function

Description

This defect occurs when you pass a narrow character string to a wide string function, or a wide character string to a narrow string function.

Misuse of narrow or wide character string raises no defect on operating systems where narrow and wide character strings have the same size.

Risk

Using a narrow character string with a wide string function, or vice versa, can result in unexpected or undefined behavior.

If you pass a wide character string to a narrow string function, you can encounter these issues:

- Data truncation. If the string contains null bytes, a copy operation using `strncpy()` can terminate early.
- Incorrect string length. `strlen()` returns the number of characters of a string up to the first null byte. A wide string can have additional characters after its first null byte.

If you pass a narrow character string to a wide string function, you can encounter this issue:

- Buffer overflow. In a copy operation using `wcsncpy()`, the destination string might have insufficient memory to store the result of the copy.

Fix

Use the narrow string functions with narrow character strings. Use the wide string functions with wide character strings.

Examples

Passing Wide Character Strings to `strncpy()`

```
#include <string.h>
#include <wchar.h>

void func(void)
{
    wchar_t wide_str1[]  = L"0123456789";
    wchar_t wide_str2[] =  L"0000000000";
    strncpy(wide_str2, wide_str1, 10);
}
```

In this example, `strncpy()` copies 10 wide characters from `wide_strt1` to `wide_str2`. If `wide_str1` contains null bytes, the copy operation can end prematurely and truncate the wide character string.

Correction — Use `wcsncpy()` to Copy Wide Character Strings

One possible correction is to use `wcsncpy()` to copy `wide_str1` to `wide_str2`.

```
#include <string.h>
#include <wchar.h>

void func(void)
{
    wchar_t wide_str1[]  = L"0123456789";
    wchar_t wide_str2[] =  L"0000000000";
    wcsncpy(wide_str2, wide_str1, 10);
}
```

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: `NARROW_WIDE_STR_MISUSE`
Impact: High
CWE ID: 135

See Also

`Array access out of bounds` | `Destination buffer overflow in string manipulation` | `Find defects (-checkers)` | `Invalid use of standard library routine` | `Invalid use of standard library string routine` | `Pointer access out of bounds` | `Unreliable cast of function pointer` | `Wrong allocated object size for cast`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Misuse of return value from nonreentrant standard function

Pointer to static buffer from previous call is used despite a subsequent call that modifies the buffer

Description

This defect occurs when these events happen in this sequence:

1. You point to the buffer returned from a nonreentrant standard function such as `getenv` or `setlocale`.

   ```
   user = getenv("USER");
   ```

2. You call that nonreentrant standard function again.

   ```
   user2 = getenv("USER2");
   ```

3. You use or dereference the pointer from the first step expecting the buffer to remain unmodified since that step. In the meantime, the call in the second step has modified the buffer.

   For instance:

   ```
   var=*user;
   ```

In some cases, the defect might appear even if you do not call the `getenv` function a second time but simply return the pointer. For instance:

```
char* func() {
     user=getenv("USER");
     .
     .
     return user;
}
```

For information on which functions are covered by this defect, see documentation on nonreentrant standard functions.

Risk

The C Standard allows nonreentrant functions such as `getenv` to return a pointer to a *static* buffer. Because the buffer is static, a second call to `getenv` modifies the buffer. If you continue to use the pointer returned from the first call past the second call, you can see unexpected results. The buffer that it points to no longer has values from the first call.

The defect appears even if you do not call `getenv` a second time but simply return the pointer. The reason is that someone calling your function might use the returned pointer *after* a second call to `getenv`. By returning the pointer from your call to `getenv`, you make your function unsafe to use.

The same rationale is true for other nonreentrant functions covered by this defect.

Fix

After the first call to `getenv`, make a copy of the buffer that the returned pointer points to. After the second call to `getenv`, use this copy. Even if the second call modifies the buffer, your copy is untouched.

Examples

Return from `getenv` Used After Second Call to `getenv`

```
#include <stdlib.h>
#include <string.h>

int func()
{
    int result = 0;

    char *home = getenv("HOME");   /* First call */
    if (home != NULL) {
        char *user = NULL;
        char *user_name_from_home = strrchr(home, '/');

        if (user_name_from_home != NULL) {
            user = getenv("USER");   /* Second call */
            if ((user != NULL) &&
                (strcmp(user, user_name_from_home) == 0))
            {
                result = 1;
            }
        }
    }
    return result;
}
```

In this example, the pointer `user_name_from_home` is derived from the pointer `home`. `home` points to the buffer returned from the first call to `getenv`. Therefore, `user_name_from_home` points to a location in the same buffer.

After the second call to `getenv`, the buffer is modified. If you continue to use `user_name_from_home`, you can get unexpected results.

Correction — Make Copy of Buffer Before Second Call

If you want to access the buffer from the first call to `getenv` past the second call, make a copy of the buffer after the first call. One possible correction is to use the `strdup` function to make the copy.

```
#include <stdlib.h>
#include <string.h>

int func()
{
    int result = 0;

    char *home = getenv("HOME");
    if (home != NULL) {
        char *user = NULL;
        char *user_name_from_home = strrchr(home, '/');
```

```
        if (user_name_from_home != NULL) {
            /* Make copy before second call */
            char *saved_user_name_from_home = strdup(user_name_from_home);
            if (saved_user_name_from_home != NULL) {
                user = getenv("USER");
                if ((user != NULL) &&
                    (strcmp(user, saved_user_name_from_home) == 0))
                {
                    result = 1;
                }
                free(saved_user_name_from_home);
            }
        }
    }
    return result;
}
```

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: NON_REENTRANT_STD_RETURN
Impact: High

See Also

Find defects (-checkers)|Modification of internal buffer returned from nonreentrant standard function|Use of obsolete standard function

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Misuse of sign-extended character value

Data type conversion with sign extension causes unexpected behavior

Description

This defect occurs when you convert a signed or plain `char` variable containing possible negative values to a wider integer data type (or perform an arithmetic operation that does the conversion) and then use the resulting value in one of these ways:

- For comparison with `EOF` (using == or `!=`)
- As array index
- As argument to a character-handling function in `ctype.h`, for instance, `isalpha()` or `isdigit()`

If you convert a signed `char` variable with a negative value to a wider type such as `int`, the sign bit is preserved (sign extension). This can lead to specific problems even in situations where you think you have accounted for the sign bit.

For instance, the signed `char` value of -1 can represent the character `EOF` (end-of-file), which is an invalid character. Suppose a `char` variable `var` acquires this value. If you treat `var` as a `char` variable, you might want to write special code to account for this invalid character value. However, if you perform an operation such as `var++` (involving integer promotion), it leads to the value 0, which represents a valid value `'\0'` by accident. You transitioned from an invalid to a valid value through the arithmetic operation.

Even for negative values other than -1, a conversion from signed `char` to signed `int` can lead to other issues. For instance, the signed `char` value -126 is equivalent to the `unsigned char` value 130 (corresponding to an extended character `'\202'`). If you convert the value from `char` to `int`, the sign bit is preserved. If you then cast the resulting value to `unsigned int`, you get an unexpectedly large value, 4294967170 (assuming 32-bit `int`). If your code expects the `unsigned char` value of 130 in the final `unsigned int` variable, you can see unexpected results.

The underlying cause of this issue is the sign extension during conversion to a wider type. Most architectures use two's complement representation for storing values. In this representation, the most significant bit indicates the sign of the value. When converted to a wider type, the conversion is done by copying this sign bit to all the leading bits of the wider type, so that the sign is preserved. For instance, the `char` value of -3 is represented as `11111101` (assuming 8-bit `char`). When converted to `int`, the representation is:

```
11111111 11111111 11111111  11111101
```

The value -3 is preserved in the wider type `int`. However, when converted to `unsigned int`, the value (4294967293) is no longer the same as the `unsigned char` equivalent of the original `char` value. If you are not aware of this issue, you can see unexpected results in your code.

Risk

In the following cases, Bug Finder flags use of variables after a conversion from `char` to a wider data type or an arithmetic operation that implicitly converts the variable to a wider data type:

- *If you compare the variable value with EOF*:

  A `char` value of -1 can represent the invalid character `EOF` or the valid extended character value `'\377'` (corresponding to the `unsigned char` equivalent, 255). After a `char` variable is cast to a wider type such as `int`, because of sign extension, the `char` value -1, representing one of `EOF` or `'\377'` becomes the `int` value -1, representing only `EOF`. The `unsigned char` value 255 can no longer be recovered from the `int` variable. Bug Finder flags this situation so that you can cast the variable to `unsigned char` first (or avoid the `char`-to-`int` conversion or converting operation before comparison with `EOF`). Only then, a comparison with `EOF` is meaningful. See “Sign-Extended Character Value Compared with EOF” on page 3-223.
- *If you use the variable value as an array index*:

  After a `char` variable is cast to a wider type such as `int`, because of sign extension, all negative values retain their sign. If you use the negative values directly to access an array, you cause buffer overflow/underflow. Even when you account for the negative values, the way you account for them might result in incorrect elements being read from the array. See “Sign-Extended Character Value Used as Array Index” on page 3-224.
- *If you pass the variable value as argument to a character-handling function*:

  According to the C11 standard (Section 7.4), if you supply an integer argument that cannot be represented as `unsigned char` or `EOF`, the resulting behavior is undefined. Bug Finder flags this situation because negative `char` values after conversion can no longer be represented as `unsigned char` or `EOF`. For instance, the signed `char` value -126 is equivalent to the `unsigned char` value 130, but the signed `int` value -126 cannot be represented as `unsigned char` or `EOF`.

Fix

Before conversion to a wider integer data type, cast the signed or plain `char` value explicitly to `unsigned char`.

If you use the `char` data type to not represent characters but simply as a smaller data type to save memory, your use of sign-extended `char` values might avoid the risks mentioned earlier. If so, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Sign-Extended Character Value Compared with EOF

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];

static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = *buf++;
    }
    return c;
}
```

```
void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

In this example, the function `parser` can traverse a string input `buf`. If a character in the string has the value -1, it can represent either `EOF` or the valid character value `'\377'` (corresponding to the `unsigned char` equivalent 255). When converted to the `int` variable `c`, its value becomes the integer value -1, which is always `EOF`. The later comparison with `EOF` will not detect if the value returned from `parser` is actually `EOF`.

Correction — Cast to `unsigned char` Before Conversion

One possible correction is to cast the plain `char` value to `unsigned char` before conversion to the wider `int` type. Only then can you test if the return value of `parser` is really `EOF`.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];

static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = (unsigned char)*buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

Sign-Extended Character Value Used as Array Index

```
#include <limits.h>
#include <stddef.h>
#include <stdio.h>

#define NUL '\0'
#define SOH 1    /* start of heading */
#define STX 2    /* start of text */
#define ETX 3    /* end of text */
#define EOT 4    /* end of transmission */
#define ENQ 5    /* enquiry */
#define ACK 6    /* acknowledge */

static const int ascii_table[UCHAR_MAX + 1] =
{
```

```
      [0]=NUL,[1]=SOH, [2]=STX, [3]=ETX, [4]=EOT, [5]=ENQ,[6]=ACK,
      /* ... */
      [126] = '~',
      /* ... */
      [130/*-126*/]='\202',
      /* ... */
      [255 /*-1*/]='\377'
};

int lookup_ascii_table(char c)
{
    int i;
    i = (c < 0 ? -c : c);
    return ascii_table[i];
}
```

In this example, the `char` variable `c` is converted to the `int` variable `i`. If `c` has negative values, they are converted to positive values before assignment to `i`. However, this conversion can lead to unexpected values when `i` is used as array index. For instance:

- If `c` has the value -1 representing the invalid character `EOF`, you want to probably treat this value separately. However, in this example, a value of `c` equal to -1 leads to a value of `i` equal to 1. The function `lookup_ascii_table` returns the value `ascii_table[1]` (or `SOH`) without the invalid character value `EOF` being accounted for.

  If you use the `char` data type to not represent characters but simply as a smaller data type to save memory, you need not worry about this issue.
- If `c` has a negative value, when assigned to `i`, its sign is reversed. However, if you access the elements of `ascii_table` through `i`, this sign reversal can result in unexpected values being read.

  For instance, if `c` has the value -126, `i` has the value 126. The function `lookup_ascii_table` returns the value `ascii_table[126]` (or `'~'`) but you probably expected the value `ascii_table[130]` (or `'\202'`).

Correction – Cast to `unsigned char`

To correct the issues, avoid the conversion from `char` to `int`. First, check `c` for the value `EOF`. Then, cast the value of the `char` variable `c` to `unsigned char` and use the result as array index.

```
#include <limits.h>
#include <stddef.h>
#include <stdio.h>

#define NUL '\0'
#define SOH 1    /* start of heading */
#define STX 2    /* start of text */
#define ETX 3    /* end of text */
#define EOT 4    /* end of transmission */
#define ENQ 5    /* enquiry */
#define ACK 6    /* acknowledge */

static const int ascii_table[UCHAR_MAX + 1] =
{
      [0]=NUL,[1]=SOH, [2]=STX, [3]=ETX, [4]=EOT, [5]=ENQ,[6]=ACK,
      /* ... */
```

```
      [126] = '~',
      /* ... */
      [130/*-126*/]='\202',
      /* ... */
      [255 /*-1*/]='\377'
};

int lookup_ascii_table(char c)
{
    int r = EOF;
    if (c != EOF) /* specific handling EOF, invalid character */
        r = ascii_table[(unsigned char)c]; /* cast to 'unsigned char' */
    return r;
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `CHARACTER_MISUSE`
Impact: Medium
CWE ID: 704

See Also

`Character value absorbed into EOF` | `Errno not checked` | `Find defects (-checkers)` | `Invalid use of standard library integer routine` | `Returned value of a sensitive function not checked`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Misuse of structure with flexible array member

Memory allocation ignores flexible array member

Description

This defect occurs when:

- You define an object with a flexible array member of unknown size at compilation time.
- You make an assignment between structures with a flexible array member without using `memcpy()` or a similar function.
- You use a structure with a flexible array member as an argument to a function and pass the argument by value.
- Your function returns a structure with a flexible array member.

A flexible array member has no array size specified and is the last element of a structure with at least two named members.

Risk

If the size of the flexible array member is not defined, it is ignored when allocating memory for the containing structure. Accessing such a structure has undefined behavior.

Fix

- Use `malloc()` or a similar function to allocate memory for a structure with a flexible array member.
- Use `memcpy()` or a similar function to copy a structure with a flexible array member.
- Pass a structure with a flexible array member as a function argument by pointer.

Examples

Structure Passed By Value to Function

```
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>


struct example_struct
{
    size_t num;
    int data[];
};

extern void arg_by_value(struct example_struct s);

void func(void)
{
```

```
    struct example_struct *flex_struct;
    size_t i;
    size_t array_size = 4;
    /* Dynamically allocate memory for the struct */
    flex_struct = (struct example_struct *)
        malloc(sizeof(struct example_struct) + sizeof(int) * array_size);
    if (flex_struct == NULL)
    {
        /* Handle error */
    }
    /* Initialize structure */
    flex_struct->num = array_size;
    for (i = 0; i < array_size; ++i)
    {
        flex_struct->data[i] = 0;
    }
    /* Handle structure */

    /* Argument passed by value. 'data' not
    copied to passed value. */
    arg_by_value(*flex_struct);

    /* Free dynamically allocated memory */
    free(flex_struct);
}
```

In this example, `flex_struct` is passed by value as an argument to `arg_by_value`. As a result, the flexible array member data is not copied to the passed argument.

Correction — Pass Structure by Pointer to Function

To ensure that all the members of the structure are copied to the passed argument, pass `flex_struct` to `arg_by_pointer` by pointer.

```
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>


struct example_struct
{
    size_t num;
    int data[];
};

extern void arg_by_pointer(struct example_struct *s);

void func(void)
{
    struct example_struct *flex_struct;
    size_t i;
    size_t array_size = 4;
    /* Dynamically allocate memory for the struct */
    flex_struct = (struct example_struct *)
```

```
        malloc(sizeof(struct example_struct) + sizeof(int) * array_size);
    if (flex_struct == NULL)
    {
        /* Handler error */
    }
    /* Initialize structure */
    flex_struct->num = array_size;
    for (i = 0; i < array_size; ++i)
    {
        flex_struct->data[i] = 0;
    }
    /* Handle structure */

    /* Structure passed by pointer */
    arg_by_pointer(flex_struct);

    /* Free dynamically allocated memory */
    free(flex_struct);
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `FLEXIBLE_ARRAY_MEMBER_STRUCT_MISUSE`
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Modification of internal buffer returned from nonreentrant standard function

Function attempts to modify internal buffer returned from a nonreentrant standard function

Description

This defect occurs when the following happens:

- A nonreentrant standard function returns a pointer.
- You attempt to write to the memory location that the pointer points to.

Nonreentrant standard functions that return a non `const`-qualified pointer to an internal buffer include `getenv`, `getlogin`, `crypt`, `setlocale`, `localeconv`, `strerror` and others.

Risk

Modifying the internal buffer that a nonreentrant standard function returns can cause the following issues:

- It is possible that the modification does not succeed or alters other internal data.

  For instance, `getenv` returns a pointer to an environment variable value. If you modify this value, you alter the environment of the process and corrupt other internal data.
- Even if the modification succeeds, it is possible that a subsequent call to the same standard function does not return your modified value.

  For instance, you modify the environment variable value that `getenv` returns. If another process, thread, or signal handler calls `setenv`, the modified value is overwritten. Therefore, a subsequent call to `getenv` does not return your modified value.

Fix

Avoid modifying the internal buffer using the pointer returned from the function.

Examples

Modification of `getenv` Return Value

```
#include <stdlib.h>
#include <string.h>

void printstr(const char*);

void func() {
    char* env = getenv("LANGUAGE");
    if (env != NULL) {
        strncpy(env, "C", 1);
        printstr(env);
    }
}
```

In this example, the first argument of `strncpy` is the return value from a nonreentrant standard function `getenv`. The behavior can be undefined because `strncpy` modifies this argument.

Correction - Copy Return Value of `getenv` and Modify Copy

One possible solution is to copy the return value of `getenv` and pass the copy to the `strncpy` function.

```
#include <stdlib.h>
#include <string.h>
enum {
    SIZE20 = 20
};

void printstr(const char*);

void func() {
    char* env = getenv("LANGUAGE");
    if (env != NULL) {
        char env_cp[SIZE20];
        strncpy(env_cp, env, SIZE20);
        strncpy(env_cp, "C", 1);
        printstr(env_cp);
    }
}
```

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: `WRITE_INTERNAL_BUFFER_RETURNED_FROM_STD_FUNC`
Impact: Low
CWE ID: 573, 628

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Overlapping assignment

Memory overlap between left and right sides of an assignment

Description

This defect occurs when there is a memory overlap between the left and right sides of an assignment. For instance, a variable is assigned to itself or one member of a union is assigned to another.

Risk

If the left and right sides of an assignment have memory overlap, the behavior is either redundant or undefined. For instance:

- Self-assignment such as `x=(int)(long)x;` is redundant unless `x` is `volatile`-qualified.
- Assignment of one union member to another causes undefined behavior.

  For instance, in the following code:

  - The result of the assignment `u1.a = u1.b` is undefined because `u1.b` is not initialized.
  - The result of the assignment `u2.b = u2.a` depends on the alignment and endianness of the implementation. It is not defined by C standards.

```
union {
   char a;
   int b;
}u1={'a'}, u2={'a'}; //'u1.a' and 'u2.a' are initialized

u1.a = u1.b;
u2.b = u2.a;
```

Fix

Avoid assignment between two variables that have overlapping memory.

Examples

Assignment of Union Members

```
#include <string.h>

union Data {
    int i;
    float f;
};

int main( ) {
    union Data data;
    data.i = 0;
    data.f = data.i;

    return 0;
}
```

In this example, the variables `data.i` and `data.f` are part of the same `union` and are stored in the same location. Therefore, part of their memory storage overlaps.

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: `OVERLAPPING_ASSIGN`
Impact: Low
CWE ID: 665

See Also

`Copy of overlapping memory` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Possible misuse of sizeof

Use of `sizeof` operator can cause unintended results

Description

This defect occurs when Polyspace Bug Finder detects possibly unintended results from the use of `sizeof` operator. For instance:

- You use the `sizeof` operator on an array parameter name, expecting the array size. However, the array parameter name by itself is a pointer. The `sizeof` operator returns the size of that pointer.
- You use the `sizeof` operator on an array element, expecting the array size. However, the operator returns the size of the array element.
- The size argument of certain functions such as `strncmp` or `wcsncpy` is incorrect because you used the `sizeof` operator earlier with possibly incorrect expectations. For instance:
  - In a function call `strncmp(string1, string2, num)`, `num` is obtained from an incorrect use of the `sizeof` operator on a pointer.
  - In a function call `wcsncpy(destination, source, num)`, `num` is the not the number of wide characters but a size in bytes obtained by using the `sizeof` operator. For instance, you use `wcsncpy(destination, source, sizeof(destination) - 1)` instead of `wcsncpy(destination, source, (sizeof(desintation)/sizeof(wchar_t)) - 1)`.

Risk

Incorrect use of the `sizeof` operator can cause the following issues:

- If you expect the `sizeof` operator to return array size and use the return value to constrain a loop, the number of loop runs are smaller than what you expect.
- If you use the return value of `sizeof` operator to allocate a buffer, the buffer size is smaller than what you require. Insufficient buffer can lead to resultant weaknesses such as buffer overflows.
- If you use the return value of `sizeof` operator incorrectly in a function call, the function does not behave as you expect.

Fix

Possible fixes are:

- Do not use the `sizeof` operator on an array parameter name or array element to determine array size.

  The best practice is to pass the array size as a separate function parameter and use that parameter in the function body.
- Use the `sizeof` operator carefully to determine the number argument of functions such as `strncmp` or `wcsncpy`. For instance, for wide string functions such as `wcsncpy`, use the number of wide characters as argument instead of the number of bytes.

Examples

`sizeof` Used Incorrectly to Determine Array Size

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < sizeof(a)/sizeof(int); i++)    {
        a[i] = i + 1;
    }
}
```

In this example, `sizeof(a)` returns the size of the pointer `a` and not the array size.

Correction — Determine Array Size in Another Way

One possible correction is to use another means to determine the array size.

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < MAX_SIZE; i++)    {
        a[i] = i + 1;
    }
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `SIZEOF_MISUSE`
Impact: High
CWE ID: 467

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

External Websites

Linux man page for strncmp
Linux man page for wcsncpy

Introduced in R2015b

Possibly unintended evaluation of expression because of operator precedence rules

Operator precedence rules cause unexpected evaluation order in arithmetic expression

Description

This defect occurs when an arithmetic expression result is possibly unintended because operator precedence rules dictate an evaluation order that you do not expect.

The defect highlights expressions of the form `x` *`op_1`* `y` *`op_2`* `z`. Here, *`op_1`* and *`op_2`* are operator combinations that commonly induce this error. For instance, `x == y | z`.

The checker does not flag all operator combinations. For instance, `x == y || z` is not flagged because you most likely intended to perform a logical OR between `x == y` and `z`. Specifically, the checker flags these combinations:

- `&&` and `||`: For instance, `x || y && z` or `x && y || z`.
- Assignment and bitwise operations: For instance, `x = y | z`.
- Assignment and comparison operations: For instance, `x = y != z` or `x = y > z`.
- Comparison operations: For instance, `x > y > z` (except when one of the comparisons is an equality `x == y > z`).
- Shift and numerical operation: For instance, `x << y + 2`.
- Pointer dereference and arithmetic: For instance, `*p++`.

Risk

The defect can cause the following issues:

- If you or another code reviewer reviews the code, the intended order of evaluation is not immediately clear.
- It is possible that the result of the evaluation does not meet your expectations. For instance:
  - In the operation `*p++`, it is possible that you expect the dereferenced value to be incremented. However, the pointer `p` is incremented before the dereference.
  - In the operation `(x == y | z)`, it is possible that you expect `x` to be compared with `y | z`. However, the `==` operation happens before the `|` operation.

Fix

See if the order of evaluation is what you intend. If not, apply parentheses to implement the evaluation order that you want.

For better readability of your code, it is good practice to apply parenthesis to implement an evaluation order even when operator precedence rules impose that order.

Examples

Expressions with Possibly Unintended Evaluation Order

```
int test(int a, int b, int c) {
    return(a & b == c);
}
```

In this example, the == operation happens first, followed by the & operation. If you intended the reverse order of operations, the result is not what you expect.

Correction — Parenthesis For Intended Order

One possible correction is to apply parenthesis to implement the intended evaluation order.

```
int test(int a, int b, int c) {
    return((a & b) == c);
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `OPERATOR_PRECEDENCE`
Impact: High
CWE ID: 783

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

External Websites

C++ Operator Precedence

Introduced in R2015b

Predefined macro used as an object

You use standard library macros such as `assert` and `errno` as objects

Description

This defect occurs when you use certain identifiers in a way that requires an underlying object to be present. These identifiers are defined as macros. The C Standard does not allow you to redefine them as objects. You use the identifiers in such a way that macro expansion of the identifiers cannot occur.

For instance, you refer to an external variable `errno`:

```
extern int errno;
```

However, `errno` does not occur as a variable but a macro.

The defect applies to these macros: `assert`, `errno`, `math_errhandling`, `setjmp`, `va_arg`, `va_copy`, `va_end`, and `va_start`. The checker looks for the defect only in source files (not header files).

Risk

The C11 Standard (Sec. 7.1.4) allows you to redefine most macros as objects. To access the object and not the macro in a source file, you do one of these:

- Redeclare the identifier as an external variable or function.
- For function-like macros, enclose the identifier name in parentheses.

If you try to use these strategies for macros that cannot be redefined as objects, an error occurs.

Fix

Do not use the identifiers in such a way that a macro expansion is suppressed.

- Do not redeclare the identifiers as external variables or functions.
- For function-like macros, do not enclose the macro name in parentheses.

Examples

Use of `assert` as Function

```
#include<assert.h>
typedef void (*err_handler_func)(int);

extern void demo_handle_err(err_handler_func, int);

void func(int err_code) {
    extern void assert(int);
    demo_handle_err(&(assert), err_code);
}
```

In this example, the `assert` macro is redefined as an external function. When passed as an argument to `demo_handle_err`, the identifier `assert` is enclosed in parentheses, which suppresses use of the `assert` macro.

Correction — Use `assert` as Macro

One possible correction is to directly use the `assert` macro from `assert.h`. A different implementation of the function `demo_handle_err` directly uses the `assert` macro instead of taking the address of an `assert` function.

```
#include<assert.h>
void demo_handle_err(int err_code) {
    assert(err_code == 0);
}

void func(int err_code) {
    demo_handle_err(err_code);
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `MACRO_USED_AS_OBJECT`
Impact: Low

See Also

`Find defects (-checkers)|MISRA C:2012 Rule 21.2`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Preprocessor directive in macro argument

You use a preprocessor directive in the argument to a function-like macro

Description

This defect occurs when you use a preprocessor directive in the argument to a function-like macro or a function that might be implemented as a function-like macro.

For instance, a `#ifdef` statement occurs in the argument to a `memcpy` function. The `memcpy` function might be implemented as a macro.

```
memcpy(dest, src,
    #ifdef PLATFORM1
      12
    #else
      24
    #endif
  );
```

The checker flags similar usage in `printf` and `assert`, which can also be implemented as macros.

Risk

During preprocessing, a function-like macro call is replaced by the macro body and the parameters are replaced by the arguments to the macro call (argument substitution). Suppose a macro `min()` is defined as follows.

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

When you call `min(1,2)`, it is replaced by the body `((X) < (Y) ? (X) : (Y))`. `X` and `Y` are replaced by 1 and 2.

According to the C11 Standard (Sec. 6.10.3), if the list of arguments to a function-like macro itself has preprocessing directives, the argument substitution during preprocessing is undefined.

Fix

To ensure that the argument substitution happens in an unambiguous manner, use the preprocessor directives outside the function-like macro.

For instance, to execute `memcpy` with different arguments based on a `#ifdef` directive, call `memcpy` multiple times within the `#ifdef` directive branches.

```
#ifdef PLATFORM1
    memcpy(dest, src, 12);
#else
    memcpy(dest, src, 24);
#endif
```

Examples

Directives in Function-Like Macros

```
#include <stdio.h>

#define print(A) printf(#A)

void func(void) {
    print(
#ifdef SW
          "Message 1"
#else
          "Message 2"
#endif
         );
}
```

In this example, the preprocessor directives `#ifdef` and `#endif` occur in the argument to the function-like macro `print()`.

Correction — Use Directives Outside Macro

One possible correction is to use the function-like macro multiple times in the branches of the `#ifdef` directive.

```
#include <stdio.h>

#define print(A) printf(#A)

void func(void) {
#ifdef SW
        print("Message 1");
#else
        print("Message 2");
#endif
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `PRE_DIRECTIVE_MACRO_ARG`
Impact: Low

See Also

`Find defects (-checkers)|MISRA C:2012 Rule 20.6`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Qualifier removed in conversion

Variable qualifier is lost during conversion

Description

This defect occurs during a pointer conversion when one pointer has a qualifier and the other does not. For example, when converting from a `const int*` to an `int*`, the conversion removes the `const` qualifier.

This defect applies only for projects in C.

Risk

Qualifiers such as `const` or `volatile` in a pointer declaration:

```
const int* ptr;
```

imply that the underlying object is `const` or `volatile`. These qualifiers act as instructions to the compiler. For instance, a `const` object is not supposed to be modified in the code and a `volatile` object is not supposed to be optimized away by the compiler.

If a second pointer points to the same object but does not use the same qualifier, the qualifier on the first pointer is no longer valid. For instance, if a `const int*` pointer and an `int*` pointer point to the same object, you can modify the object through the second pointer and violate the contract implied by the `const` qualifier in the first pointer.

Fix

If you intend to convert from one pointer to another, declare both pointers with the same qualifiers.

Examples

Cast of Character Pointers

```
void implicit_cast(void) {
    const char  cc, *pcc = &cc;
    char * quo;

    quo = &cc;
    quo = pcc;

    read(quo);
}
```

During the assignment to the character `q`, the variables, `cc` and `pcc`, are converted from `const char` to `char`. The `const` qualifier is removed during the conversion causing a defect.

Correction — Add Qualifiers

One possible correction is to add the same qualifiers to the new variables. In this example, changing `q` to a `const char` fixes the defect.

```
void implicit_cast(void) {
    const char  cc, *pcc = &cc;
```

```
    const char * quo;

    quo = &cc;
    quo = pcc;

    read(quo);
}
```

Correction — Remove Qualifiers

One possible correction is to remove the qualifiers in the converted variable. In this example, removing the `const` qualifier from the `cc` and `pcc` initialization fixes the defect.

```
void implicit_basic_cast(void) {
    char  cc, *pcc = &cc;
    char * quo;

    quo = &cc;
    quo = pcc;

    read(quo);
}
```

Check Information

Group: Programming
Language: C
Default: Off
Command-Line Syntax: `QUALIFIER_MISMATCH`
Impact: Low
CWE ID: 704

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Return from computational exception signal handler

Undefined behavior when signal handler returns normally from program error

Description

This defect occurs when a signal handler returns after catching a computational exception signal `SIGFPE`, `SIGILL`, or `SIGSEGV`.

Risk

A signal handler that returns normally from a computational exception is undefined behavior. Even if the handler attempts to fix the error that triggered the signal, the program can behave unexpectedly.

Fix

Check the validity of the values of your variables before the computation to avoid using a signal handler to catch exceptions. If you cannot avoid a handler to catch computation exception signals, call `abort()`, `quick_exit()`, or `_Exit()` in the handler to stop the program.

Examples

Signal Handler Return from Division by Zero

```
#include <errno.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>

static volatile sig_atomic_t denom;
/* Declare signal handler to catch division by zero
computation error. */
void sig_handler(int s)
{
    int s0 = s;
    if (denom == 0)
    {
        denom = 1;
    }
    /* Normal return from computation exception
    signal */
    return;
}


long func(int v)
{
    denom = (sig_atomic_t)v;

        if (signal(SIGFPE, sig_handler) == SIG_ERR)
        {
            /* Handle error */
```

```
        }

    long result = 100 / (long)denom;
    return result;
}
```

In this example, `sig_handler` is declared to handle a division by zero computation error. The handler changes the value of `denom` if it is zero and returns, which is undefined behavior.

Correction — Call `abort()` to Terminate Program

After catching a computational exception, call `abort()` from `sig_handler` to exit the program without further error.

```
#include <errno.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>

static volatile sig_atomic_t denom;
/* Declare signal handler to catch division by zero
computation error. */

void sig_handler(int s)
{
    int s0 = s;
    /* call to abort() to exit the program */
    abort();
}

long func(int v)
{
    denom = (sig_atomic_t)v;

        if (signal(SIGFPE, sig_handler) == SIG_ERR)
        {
            /* Handle error */
        }

    long result = 100 / (long)denom;
    return result;
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `SIG_HANDLER_COMP_EXCP_RETURN`
Impact: Low
CWE ID: 387

See Also

`Find defects (-checkers)` | `Function called from signal handler not asynchronous-safe` | `Function called from signal handler not asynchronous-safe (strict)` | `Signal call from within signal handler`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Shared data access within signal handler

Access or modification of shared data causes inconsistent state

Description

This defect occurs when you access or modify a shared object inside a signal handler.

Risk

When you define a signal handler function to access or modify a shared object, the handler accesses or modifies the shared object when it receives a signal. If another function is already accessing the shared object, that function causes a race condition and can leave the data in an inconsistent state.

Fix

To access or modify shared objects inside a signal handler, check that the objects are lock-free atomic, or, if they are integers, declare them as `volatile sig_atomic_t`.

Examples

`int` Variable Access in Signal Handler

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* declare global variable. */
int e_flag;

void sig_handler(int signum)
{
    /* Signal handler accesses variable that is not
    of type volatile sig_atomic_t. */
    e_flag = signum;
}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

In this example, `sig_handler` accesses `e_flag`, a variable of type `int`. A concurrent access by another function can leave `e_flag` in an inconsistent state.

Correction — Declare Variable of Type `volatile sig_atomic_t`

Before you access a shared variable from a signal handler, declare the variable with type `volatile sig_atomic_t` instead of `int`. You can safely access variables of this type asynchronously.

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* Declare variable of type volatile sig_atomic_t. */
volatile sig_atomic_t e_flag;
void sig_handler(int signum)
{
    /* Use variable of proper type inside signal handler. */
    e_flag = signum;

}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `SIG_HANDLER_SHARED_OBJECT`
Impact: Medium
CWE ID: 364, 413

See Also

`Find defects (-checkers)` | `Function called from signal handler not asynchronous-safe` | `Signal call from within signal handler`

Topics

“Interpret Polyspace Bug Finder Results”

“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Side effect in arguments to unsafe macro

Macro contains arguments that can be evaluated multiple times or not evaluated

Description

This defect occurs when you call an unsafe macro with an expression that has a side effect.

- *Unsafe macro*: When expanded, an unsafe macro evaluates its arguments multiple times or does not evaluate its argument at all.

  For instance, the `ABS` macro evaluates its argument `x` twice.

  ```
  #define ABS(x) (((x) < 0) ? -(x) : (x))
  ```

- *Side effect*: When evaluated, an expression with a side effect modifies at least one of the variables in the expression.

  For instance, `++n` modifies `n`, but `n+1` does not modify `n`.

  The checker does not consider side effects in nested macros. The checker also does not consider function calls or volatile variable access as side effects.

Risk

If you call an unsafe macro with an expression that has a side effect, the expression is evaluated multiple times or not evaluated at all. The side effect can occur multiple times or not occur at all, causing unexpected behavior.

For instance, in the call `MACRO(++n)`, you expect only one increment of the variable `n`. If `MACRO` is an unsafe macro, the increment happens more than once or does not happen at all.

The checker flags expressions with side effects in the `assert` macro because the `assert` macro is disabled in non-debug mode. To compile in non-debug mode, you define the `NDEBUG` macro during compilation. For instance, in GCC, you use the flag `-DNDEBUG`.

Fix

Evaluate the expression with a side effect in a separate statement, and then use the result as a macro argument.

For instance, instead of:

```
MACRO(++n);
```

perform the operation in two steps:

```
++n;
MACRO(n);
```

Alternatively, use an inline function instead of a macro. Pass the expression with side effect as argument to the inline function.

The checker considers modifications of a local variable defined only in the block scope of a macro body as a side effect. This defect cannot happen since the variable is visible only in the macro body. If you see a defect of this kind, ignore the defect.

Examples

Macro Argument with Side Effects

```
#define ABS(x) (((x) < 0) ? -(x) : (x))

void func(int n) {
  /* Validate that n is within the desired range */
  int m = ABS(++n);

  /* ... */
}
```

In this example, the ABS macro evaluates its argument twice. The second evaluation can result in an unintended increment.

Correction — Separate Evaluation of Expression from Macro Usage

One possible correction is to first perform the increment, and then pass the result to the macro.

```
#define ABS(x) (((x) < 0) ? -(x) : (x))

void func(int n) {
  /* Validate that n is within the desired range */
  ++n;
  int m = ABS(n);

  /* ... */
}
```

Correction — Evaluate Expression in Inline Function

Another possible correction is to evaluate the expression in an inline function.

```
static inline int iabs(int x) {
  return (((x) < 0) ? -(x) : (x));
}

void func(int n) {
  /* Validate that n is within the desired range */

int m = iabs(++n);

  /* ... */
}
```

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: SIDE_EFFECT_IN_UNSAFE_MACRO_ARG
Impact: Medium

See Also

`Find defects (-checkers)|MISRA C:2012 Rule 13.2|MISRA C:2012 Rule 13.3|MISRA C:2012 Rule 13.4|Side effect of expression ignored|Stream argument with possibly unintended side effects`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Side effect of expression ignored

`sizeof`, `_Alignof`, or `_Generic` operates on expression with side effect

Description

This defect occurs when the `sizeof`, `_Alignof`, or `_Generic` operator operates on an expression with a side effect. When evaluated, an expression with side effect modifies at least one of the variables in the expression.

For instance, the defect checker does not flag `sizeof(n+1)` because `n+1` does not modify `n`. The checker flags `sizeof(n++)` because `n++` is intended to modify `n`.

The check also applies to the C++ operator `alignof` and its C extensions, `__alignof__` and `__typeof__`.

Risk

The expression in a `_Alignof` or `_Generic` operator is not evaluated. The expression in a `sizeof` operator is evaluated only if it is required for calculating the size of a variable-length array, for instance, `sizeof(a[n++])`.

When an expression with a side effect is not evaluated, the variable modification from the side effect does not happen. If you rely on the modification, you can see unexpected results.

Fix

Evaluate the expression with a side effect in a separate statement, and then use the result in a `sizeof`, `_Alignof`, or `_Generic` operator.

For instance, instead of:

```
a = sizeof(n++);
```

perform the operation in two steps:

```
n++;
a = sizeof(n);
```

The checker considers a function call as an expression with a side effect. Even if the function does not have side effects now, it might have side effects on later additions. The code is more maintainable if you call the function outside the `sizeof` operator.

Examples

Increment Operator in `sizeof`

```
#include <stdio.h>

void func(void) {
    unsigned int a = 1U;
    unsigned int b = (unsigned int)sizeof(++a);
    printf ("%u, %u\n", a, b);
}
```

In this example, `sizeof` operates on `++a`, which is intended to modify `a`. Because the expression is not evaluated, the modification does not happen. The `printf` statement shows that `a` still has the value `1`.

Correction — Perform Increment Outside `sizeof`

One possible correction is to perform the increment first, and then provide the result to the `sizeof` operator.

```
#include <stdio.h>

void func(void) {
    unsigned int a = 1U;
    ++a;
    unsigned int b = (unsigned int)sizeof (a);
    printf ("%u, %u\n", a, b);
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `SIDE_EFFECT_IGNORED`
Impact: Low

See Also

`Find defects (-checkers)|MISRA C:2012 Rule 13.6|Redundant expression in sizeof operand`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Signal call from within signal handler

Nonpersistent signal handler calling `signal()` in Windows system causes race condition

Description

This defect occurs when you call `signal()` from a nonpersistent signal handler on a Windows platform.

Risk

A nonpersistent signal handler is reset after catching a signal. The handler does not catch subsequent signals unless the handler is reestablished by calling `signal()`. A nonpersistent signal handler on a Windows platform is reset to `SIG_DFL`. If another signal interrupts the execution of the handler, that signal can cause a race condition between `SIG_DFL` and the existing signal handler. A call to `signal()` can also result in an infinite loop inside the handler.

Fix

Do not call `signal()` from a signal handler on Windows platforms.

Examples

`signal()` Called from Signal Handler

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;

void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;

    /* Call signal() to reestablish sig_handler
    upon receiving SIG_ERR. */

    if (signal(s0, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
}

void func(void)
{
        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */
```

```
        }
  /* more code */
}
```

In this example, the definition of `sig_handler()` includes a call to `signal()` when the handler catches `SIG_ERR`. On Windows platforms, signal handlers are nonpersistent. This code can result in a race condition.

Correction — Do Not Call `signal()` from Signal Handler

If your code requires the use of a persistent signal handler on a Windows platform, use a persistent signal handler after performing a thorough risk analysis.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;


void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;
    /* No call to signal() */
}

int main(void)
{

        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */

        }
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `SIG_HANDLER_CALLING_SIGNAL`
Impact: Medium
CWE ID: 387, 474

See Also

`Find defects (-checkers)` | `Function called from signal handler not asynchronous-safe` | `Return from computational exception signal handler` | `Shared data access within signal handler`

Topics
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Standard function call with incorrect arguments

Argument to a standard function does not meet requirements for use in the function

Description

This defect occurs when the arguments to certain standard functions do not meet the requirements for their use in the functions.

For instance, the arguments to these functions can be invalid in the following ways.

Function Type	Situation	Risk	Fix
String manipulation functions such as `strlen` and `strcpy`	The pointer arguments do not point to a `NULL`-terminated string.	The behavior of the function is undefined.	Pass a `NULL`-terminated string to string manipulation functions.
File handling functions in `stdio.h` such as `fputc` and `fread`	The `FILE*` pointer argument can have the value `NULL`.	The behavior of the function is undefined.	Test the `FILE*` pointer for `NULL` before using it as function argument.
File handling functions in `unistd.h` such as `lseek` and `read`	The file descriptor argument can be -1.	The behavior of the function is undefined. Most implementations of the `open` function return a file descriptor value of -1. In addition, they set `errno` to indicate that an error has occurred when opening a file.	Test the return value of the `open` function for -1 before using it as argument for `read` or `lseek`. If the return value is -1, check the value of `errno` to see which error has occurred.
	The file descriptor argument represents a closed file descriptor.	The behavior of the function is undefined.	Close the file descriptor only after you have completely finished using it. Alternatively, reopen the file descriptor before using it as function argument.
Directory name generation functions such as `mkdtemp` and `mkstemps`	The last six characters of the string template are not `XXXXXX`.	The function replaces the last six characters with a string that makes the file name unique. If the last six characters are not `XXXXXX`, the function cannot generate a unique enough directory name.	Test if the last six characters of a string are `XXXXXX` before using the string as function argument.

Function Type	Situation	Risk	Fix
Functions related to environment variables such as `getenv` and `setenv`	The string argument is `""`.	The behavior is implementation-defined.	Test the string argument for `""` before using it as `getenv` or `setenv` argument.
	The string argument terminates with an equal sign, =. For instance, `"C="` instead of `"C"`.	The behavior is implementation-defined.	Do not terminate the string argument with =.
String handling functions such as `strtok` and `strstr`	• `strtok`: The delimiter argument is `""`. • `strstr`: The search string argument is `""`.	Some implementations do not handle these edge cases.	Test the string for `""` before using it as function argument.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

`NULL` Pointer Passed as `strnlen` Argument

```
#include <string.h>
#include <stdlib.h>

enum {
    SIZE10 = 10,
    SIZE20 = 20
};

int func() {
    char* s = NULL;
    return strnlen(s, SIZE20);
}
```

In this example, a `NULL` pointer is passed as `strnlen` argument instead of a `NULL`-terminated string.

Before running analysis on the code, specify a GNU compiler. See `Compiler (-compiler)`.

Correction — Pass `NULL`-terminated String

Pass a `NULL`-terminated string as the first argument of `strnlen`.

```
#include <string.h>
#include <stdlib.h>

enum {
    SIZE10 = 10,
    SIZE20 = 20
};

int func() {
    char* s = "";
    return strnlen(s, SIZE20);
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `STD_FUNC_ARG_MISMATCH`
Impact: Medium
CWE ID: 628, 685, 686, 687, 690, 910

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Stream argument with possibly unintended side effects

Stream argument side effects occur more than once

Description

This defect occurs when you call `getc()`, `putc()`, `getwc()`, or `putwc()` with a stream argument that has side effects.

Stream argument with possibly unintended side effects considers the following as stream side effects:

- Any assignment of a variable of a stream, such as `FILE *`, or any assignment of a variable of a deeper stream type, such as an array of `FILE *`.
- Any call to a function that manipulates a stream or a deeper stream type.

The number of defects raised corresponds to the number of side effects detected. When a stream argument is evaluated multiple times in a function implemented as a macro, a defect is raised for each evaluation that has a side effect.

A defect is also raised on functions that are not implemented as macros but that can be implemented as macros on another operating system.

Risk

If the function is implemented as an unsafe macro, the stream argument can be evaluated more than once, and the stream side effect happens multiple times. For instance, a stream argument calling `fopen()` might open the same file multiple times, which is unspecified behavior.

Fix

To ensure that the side effect of a stream happens only once, use a separate statement for the stream argument.

Examples

Stream Argument of `getc()` Has Side Effect `fopen()`

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

#define fatal_error() abort()

const char* myfile = "my_file.log";

void func(void)
{
    int c;
    FILE* fptr;
    /* getc() has stream argument fptr with
```

```
    * 2 side effects: call to fopen(), and assignment
    * of fptr
    */
    c = getc(fptr = fopen(myfile, "r"));
    if (c == EOF) {
        /* Handle error */
        (void)fclose(fptr);
        fatal_error();
    }
    if (fclose(fptr) == EOF) {
        /* Handle error */
        fatal_error();
    }
}

void main(void)
{
    func();

}
```

In this example, `getc()` is called with stream argument `fptr`. The stream argument has two side effects: the call to `fopen()` and the assignment of `fptr`. If `getc()` is implemented as an unsafe macro, the side effects happen multiple times.

Correction — Use Separate Statement for `fopen()`

One possible correction is to use a separate statement for `fopen()`. The call to `fopen()` and the assignment of `fptr` happen in this statement so there are no side effects when you pass `fptr` to `getc()`.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

#define fatal_error() abort()


const char* myfile = "my_file.log";

void func(void)
{
    int c;
    FILE* fptr;

    /* Separate statement for fopen()
    * before call to getc()
    */
    fptr = fopen(myfile, "r");
    if (fptr == NULL) {
        /* Handle error */
        fatal_error();
    }
    c = getc(fptr);
    if (c == EOF) {
        /* Handle error */
        (void)fclose(fptr);
        fatal_error();
```

```
    }
    if (fclose(fptr) == EOF) {
        /* Handle error */
        fatal_error();
    }
}

void main(void)
{
    func();

}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: STREAM_WITH_SIDE_EFFECT
Impact: Low

See Also

`Find defects (-checkers)` | `Opening previously opened resource` | `Returned value of a sensitive function not checked` | `Standard function call with incorrect arguments`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Too many va_arg calls for current argument list

Number of calls to `va_arg` exceeds number of arguments passed to variadic function

Description

This defect occurs when the number of calls to `va_arg` exceeds the number of arguments passed to the corresponding variadic function. The analysis raises a defect only when the variadic function is called.

Too many va_arg calls for current argument list does not raise a defect when:

- The number of calls to `va_arg` inside the variadic function is indeterminate. For example, if the calls are from an external source.
- The `va_list` used in `va_arg` is invalid.

Risk

When you call `va_arg` and there is no next argument available in `va_list`, the behavior is undefined. The call to `va_arg` might corrupt data or return an unexpected result.

Fix

Ensure that you pass the correct number of arguments to the variadic function.

Examples

No Argument Available When Calling `va_arg`

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/
int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
        if (count > 0) {
/* No further argument available
* in va_list when calling va_arg
*/

            result += va_arg(ap, int);
        }
    }
    va_end(ap);
    return result;
```

```
}

void func(void) {

    (void)variadic_func(2, 100);

}
```

In this example, the named argument and only one variadic argument are passed to `variadic_func()` when it is called inside `func()`. On the second call to `va_arg`, no further variadic argument is available in `ap` and the behavior is undefined.

Correction — Pass Correct Number of Arguments to Variadic Function

One possible correction is to ensure that you pass the correct number of arguments to the variadic function.

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/

int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
        if (count > 0) {

/* The correct number of arguments is
* passed to va_list when variadic_func()
* is called inside func()
*/
            result += va_arg(ap, int);
        }
    }
    va_end(ap);
    return result;
}

void func(void) {

    (void)variadic_func(2, 100, 200);

}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `TOO_MANY_VA_ARG_CALLS`

Impact: Medium
CWE ID: 685

See Also

`Find defects (-checkers)` | `Incorrect data type passed to va_arg` | `Invalid va_list argument`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Typedef mismatch

Mismatch between `typedef` statements

Description

This defect occurs when `typedef` statements lead to conflicting underlying types for one of these data types:

- `size_t`
- `ssize_t`
- `wchar_t`
- `ptrdiff_t`

Risk

If you change the underlying type of `size_t`, `ssize_t`, `wchar_t`, or `ptrdiff_t`, you have inconsistent definitions of the same type. Compilation units with different include paths can potentially use different-sized types causing conflicts in your program.

For example, say that you define a function in one compilation unit that redefines `size_t` as unsigned long. But in another compilation unit that uses the `size_t` definition from `<stddef.h>`, you use the same function as an `extern` declaration. Your program will encounter a mismatch between the function declaration and function definition.

Fix

Use consistent type definitions. For example:

- Remove custom type definitions for these fundamental types. Only use system definitions.
- Use the same size for all compilation units. Move your `typedef` to a shared header file.

Examples

Two Definitions of `size_t`

`file1.c`:

```
#include <stddef.h>

void func1()
{
    size_t var = 0;
    /*... more code ... */
}
```

`file2.c`:

```
typedef unsigned char size_t;

void func2()
{
```

```
    size_t var = 0;
    /*... more code ... */
}
```

In this example, Polyspace flags the definition of `size_t` in `file2.c` as a defect. This definition is a `typedef` mismatch because another file in your project, `file1.c`, includes `stddef.h`, which defines `size_t` as unsigned long.

Correction — Use System Definition

One possible correction is to use the system definition of `size_t` in `stddef.h` to avoid conflicting type definitions.

`file1.c`:

```
#include <stddef.h>

void func1()
{
    size_t var = 0;
    /*... more code ... */
}
```

`file2.c`:

```
#include <stddef.h>

void func2()
{
    size_t var = 0;
    /*... more code ... */
}
```

Correction — Use Shared Header File

One possible correction is to use a shared header file to store your type definition that gets included in both files.

`types.h`:

```
typedef unsigned char size_t;
```

`file1.c`:

```
#include "types.h"

void func1()
{
    size_t var = 0;
    /*... more code ... */
}
```

`file2.c`:

```
#include "types.h"

void func2()
{
    size_t var = 0;
```

```
    /*... more code ... */
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `TYPEDEF_MISMATCH`
Impact: High

See Also

`Declaration mismatch` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2016b

Universal character name from token concatenation

You create a universal character name by joining tokens with ## operator

Description

This defect occurs when two preprocessing tokens joined with a ## operator create a universal character name. A universal character name begins with `\u` or `\U` followed by hexadecimal digits. It represents a character not found in the basic character set.

For instance, you form the character `\u0401` by joining two tokens:

```
#define assign(uc1, uc2, val) uc1##uc2 = val
...
assign(\u04, 01, 4);
```

Risk

The C11 Standard (Sec. 5.1.1.2) states that if a universal character name is formed by token concatenation, the behavior is undefined.

Fix

Use the universal character name directly instead of producing it through token concatenation.

Examples

Universal Character Name from Token Concatenation

```
#define assign(uc1, uc2, val) uc1##uc2 = val

int func(void) {
    int \u0401 = 0;
    assign(\u04, 01, 4);
    return \u0401;
}
```

In this example, the `assign` macro, when expanded, joins the two tokens `\u04` and `01` to form the universal character name `\u0401`.

Correction — Use Universal Character Name Directly

One possible correction is to use the universal character name `\u0401` directly. The correction redefines the `assign` macro so that it does not join tokens.

```
#define assign(ucn, val) ucn = val

int func(void) {
    int \u0401 = 0;
    assign(\u0401, 4);
    return \u0401;
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: PRE_UCNAME_JOIN_TOKENS
Impact: Low

See Also

`Find defects (-checkers)|MISRA C:2012 Rule 20.10`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Unnamed namespace in header file

Header file contains unnamed namespace leading to multiple definitions

Description

This defect occurs when an unnamed namespace is used in a header file, which can lead to multiple definitions of objects in the namespace.

Risk

According to the C++ standard, names in an unnamed namespace, for instance, `aVar`:

```
namespace {
   int aVar;
}
```

have internal linkage by default. If a header file contains an unnamed namespace, each translation unit with a source file that `#include`-s the header file defines its own instance of objects in the namespace. The multiple definitions are probably not what you intended and can lead to unexpected results, unwanted excess memory usage, or inadvertently violating the one-definition rule.

Fix

Specify names for namespaces in header files or avoid using namespaces in header files.

Examples

Unexpected Results from Unnamed Namespaces in Header Files

Header File: `aHeader.h`

```
namespace {
   int aVar;
}
```

First source file: `aSource.cpp`

```
#include "aHeader.h"
#include <iostream>

void setVar(int arg) {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = arg;
    std::cout << "Value set at: " << aVar << std::endl;
}
```

Second source file: `anotherSource.cpp`

```
#include "aHeader.h"
#include <iostream>

extern void setVar(int);
```

```
void resetVar() {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = 0;
    std::cout << "Value set at: 0" << std::endl;
}

void main() {
    setVar(1);
    resetVar();
}
```

In this example, the unnamed namespace leads to two definitions of `aVar` in the translation unit from `aSource.cpp` and the translation unit from `anotherSource.cpp`. The two definitions lead to possible unexpected output:

```
Current value: 0
Value set at: 1
Current value: 0
Value set at: 0
```

Correction – Avoid the Unnamed Namespace

One possible correction is to avoid a namespace in the header file.

Header File: `aHeader.h`

```
extern int aVar;
```

First source file: `aSource.cpp`

```
#include "aHeader.h"
#include <iostream>

void setVar(int arg) {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = arg;
    std::cout << "Value set at: " << aVar << std::endl;
}
```

Second source file: `anotherSource.cpp`

```
#include "aHeader.h"
#include <iostream>

extern void setVar(int);
int aVar;

void resetVar() {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = 0;
    std::cout << "Value set at: 0" << std::endl;
}

void main() {
    setVar(1);
    resetVar();
}
```

You now see the expected sequence in the output:

```
Current value: 0
Value set at: 1
Current value: 1
Value set at: 0
```

Result Information

Group: Programming
Language: C++
Default: On
Command-Line Syntax: `UNNAMED_NAMESPACE_IN_HEADER`
Impact: Medium

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019b

Unsafe conversion between pointer and integer

Misaligned or invalid results from conversions between pointer and integer types

Description

This defect occurs when you convert between a pointer type, such as `intptr_t`, or `uintprt_t`, and an integer type, such as `enum`, `ptrdiff_t`, or `pid_t`, or vice versa.

Risk

The mapping between pointers and integers is not always consistent with the addressing structure of the environment.

Converting from pointers to integers can create:

- Truncated or out of range integer values.
- Invalid integer types.

Converting from integers to pointers can create:

- Misaligned pointers or misaligned objects.
- Invalid pointer addresses.

Fix

Where possible, avoid pointer-to-integer or integer-to-pointer conversions. If you want to convert a `void` pointer to an integer, so that you do not change the value, use types:

- C99 — `intptr_t` or `uintptr_t`
- C90 — `size_t` or `ssize_t`

Examples

Integer to Pointer Conversions

```
unsigned int *badintptrcast(void)
{
    unsigned int *ptr0 = (unsigned int *)0xdeadbeef;
    char *ptr1 = (char *)0xdeadbeef;
    return (unsigned int *)(ptr0 - (unsigned int *)ptr1);
}
```

In this example, there are three conversions, two unsafe conversions and one safe conversion. The first conversion of `0xdeadbeef` to `unsigned int*` causes alignment issues for the pointer. The second conversion of `0xdeadbeef` to `char *` is safe because there are no alignment issues for `char`. The third conversion in the return casts `ptrdiff_t` to a pointer. This pointer might or might not point to an invalid address.

Correction — Use `intptr_t`

One possible correction is to use `intptr_t` types to store the pointer address `0xdeadbeef`. Also, you can change the second pointer to an integer offset so that there is no longer a conversion from `ptrdiff_t` to a pointer.

```
#include <stdint.h>

unsigned int *badintptrcast(void)
{
    intptr_t iptr0 = (intptr_t)0xdeadbeef;
    int offset = 0;
    return (unsigned int *)(iptr0 - offset);
}
```

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: `BAD_INT_PTR_CAST`
Impact: Medium
CWE ID: 465, 466, 587, 758

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2016b

Unsafe conversion from string to numerical value

String to number conversion without validation checks

Description

This defect occurs when you perform conversions from strings to integer or floating-point values and your conversion method does not include robust error handling.

Risk

Converting a string to numerical value can cause data loss or misinterpretation. Without validation of the conversion or error handling, your program continues with invalid values.

Fix

- Add additional checks to validate the numerical value.
- Use a more robust string-to-numeric conversion function such as `strtol`, `strtoll`, `strtoul`, or `strtoull`.

Examples

Conversion With `atoi`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int demo_check_string_not_empty(char *s)
{
    if (s != NULL)
        return strlen(s) > 0; /* check string null-terminated and not empty */
    else
        return 0;
}

int unsafestrtonumeric(char* argv1)
{
    int s = 0;
    if (demo_check_string_not_empty(argv1))
    {
        s = atoi(argv1);
    }
    return s;
}
```

In this example, `argv1` is converted to an integer with `atoi`. `atoi` does not provide errors for an invalid integer string. The conversion can fail unexpectedly.

Correction — Use `strtol` instead

One possible correction is to use `strtol` to validate the input string and the converted integer.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <limits.h>
#include <errno.h>

static int demo_check_string_not_empty(char *s)
{
    if (s != NULL)
        return strlen(s) > 0; /* check string null-terminated and not empty */
    else
        return 0;
}

int unsafestrtonumeric(char *argv1)
{
    char *c_str = argv1;
    char *end;
    long sl;

    if (demo_check_string_not_empty(c_str))
    {
        errno = 0; /* set errno for error check */
        sl = strtol(c_str, &end, 10);
        if (end == c_str)
        {
            (void)fprintf(stderr, "%s: not a decimal number\n", c_str);
        }
        else if ('\0' != *end)
        {
            (void)fprintf(stderr, "%s: extra characters: %s\n", c_str, end);
        }
        else if ((LONG_MIN == sl || LONG_MAX == sl) && ERANGE == errno)
        {
            (void)fprintf(stderr, "%s out of range of type long\n", c_str);
        }
        else if (sl > INT_MAX)
        {
            (void)fprintf(stderr, "%ld greater than INT_MAX\n", sl);
        }
        else if (sl < INT_MIN)
        {
            (void)fprintf(stderr, "%ld less than INT_MIN\n", sl);
        }
        else
        {
            return (int)sl;
        }
    }
    return 0;
}
```

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: UNSAFE_STR_TO_NUMERIC
Impact: Low

CWE ID: 20, 253, 676

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2016b

Use of indeterminate string

Use of buffer from fgets-family function

Description

This defect occurs when you do not check the validity of the buffer returned from `fgets`-family functions. The checker raises a defect when such a buffer is used as:

- An argument in standard functions that print or manipulate strings or wide strings.
- A return value.
- An argument in external functions with parameter type `const char *` or `const wchar_t *`.

Risk

If an `fgets`-family function fails, the content of its output buffer is indeterminate. Use of such a buffer has undefined behavior and can result in a program that stops working or other security vulnerabilities.

Fix

Reset the output buffer of an `fgets`-family function to a known string value when the function fails.

Examples

Output of `fgets()` Passed to External Function

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func(void) {
    char buf[SIZE20];

    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* 'buf' may contain an indeterminate string.  */
        ;
    }
    /* 'buf passed to external function */
    display_text(buf);
}
```

In this example, the output `buf` is passed to the external function `display_text()`, but its value is not reset if `fgets()` fails.

Correction — Reset `fgets()` Output on Failure

If `fgets()` fails, reset `buf` to a known value before you pass it to an external function.

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func1(void) {
    char buf[SIZE20];
    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* value of 'buf' reset after fgets() failure. */
        buf[0] = '\0';
    }
    /* 'buf' passed to external function */
    display_text(buf);
}
```

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: `INDETERMINATE_STRING`
Impact: Medium

See Also

`Find defects (-checkers)` | `Invalid use of standard library string routine` | `Returned value of a sensitive function not checked` | `Use of dangerous standard function`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Use of memset with size argument zero

Size argument of function in `memset` family is zero

Description

This defect occurs when you call a function in the `memset` family with size argument zero. Functions include `memset`, `wmemset`, `bzero`, `SecureZeroMemory`, `RtlSecureZeroMemory`, and so on.

Risk

`void *memset (void *ptr, int value, size_t num)` fills the first `num` bytes of the memory block that `ptr` points to with the specified `value`. A zero value of `num` renders the call to `memset` redundant. The memory that `ptr` points to:

- Remains uninitialized, if not previously initialized.
- Is not cleared and can contain sensitive data, if previously initialized.

Fix

Determine if the zero size argument occurs because of a previous error in your code. Fix the error.

Examples

Zero Size Argument of `memset`

```
#include <stdio.h>
#include <string.h>

void func (unsigned int size)
{
    char str[] = "Buffer to be filled.";
    memset (str,'-',size);
    puts (str);
}

void calling_func(void) {
    unsigned int buf_size=0;
    func(buf_size);
}
```

In this example, the argument `size` of `memset` is zero.

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: `MEMSET_INVALID_SIZE`
Impact: Medium
CWE ID: 665

See Also

`Call to memset with unintended value` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Variable length array with nonpositive size

Size of variable-length array is zero or negative

Description

This defect occurs when size of a variable-length array is zero or negative.

Risk

If the size of a variable-length array is zero or negative, unexpected behavior can occur, such as stack overflow.

Fix

When you declare a variable-length array as a local variable in a function:

- If you use a function parameter as the array size, check that the parameter is positive.
- If you use the result of a computation on a function parameter as the array size, check that the result is positive.

You can place a test for positive value either before the function call or the array declaration in the function body.

Examples

Nonpositive Array Size

```
int input(void);

void add_scalar(int n, int m) {
    int r=0;
    int arr[m][n];
    for (int i=0; i<m; i++) {
        for (int j=0; j<n; j++) {
            arr[i][j] = input();
            r += arr[i][j];
        }
    }
}

void main() {
    add_scalar(2,2);
    add_scalar(-1,2);
    add_scalar(2,0);
}
```

In this example, the second and third calls to `add_scalar` result in a negative and zero size of `arr`.

Correction — Make Array Size Positive

One possible correction is fix or remove calls that result in a nonpositive array size.

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `NON_POSITIVE_VLA_SIZE`
Impact: High
CWE ID: 687

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Writing to const qualified object

Object declared with a `const` qualifier is modified

Description

This defect occurs when you do one of the following:

- Use a `const`-qualified object as the destination of an assignment.
- Pass a `const`-qualified object to a function that modifies the argument.

For instance, the defect can occur in the following situations:

- You pass a `const`-qualified object as first argument of one of the following functions:
  - `mkstemp`
  - `mkostemp`
  - `mkostemps`
  - `mkdtemp`
- You pass a `const`-qualified object as the destination argument of one of the following functions:
  - `strcpy`
  - `strncpy`
  - `strcat`
  - `memset`
- You perform a write operation on a `const`-qualified object.

Risk

The risk depends upon the modifications made to the `const`-qualified object.

Situation	Risk
Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on.	These functions replace the last six characters of their first argument with a string. Therefore, they expect a modifiable `char` array as their first argument.
Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on.	These functions modify their destination argument. Therefore, they expect a modifiable `char` array as their destination argument.
Writing to the object	The `const` qualifier implies an agreement that the value of the object will not be modified. By writing to a `const`-qualified object, you break the agreement. The result of the operation is undefined.

Fix

The fix depends on the modification made to the `const`-qualified object.

Situation	Fix
Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on.	Pass a non-`const` object as first argument of the function.
Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on.	Pass a non-`const` object as destination argument of the function.
Writing to the object	Perform the write operation on a non-`const` object.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Writing to `const`-Qualified Object

```
#include <string.h>

const char* buffer = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

In this example, because `buffer` is `const`-qualified, `strchr(buffer,'X')` returns a `const`-qualified `char*` pointer. When this `char*` pointer is used as the destination argument of `strcpy`, a **Writing to const qualified object** error appears.

Correction — Copy `const`-Qualified Object to Non-`const` Object

One possible correction is to assign the constant string to a non-`const` object and use the non-`const` object as destination argument of `strchr`.

```
#include <string.h>

char buffer[] = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: `CONSTANT_OBJECT_WRITE`
Impact: High
CWE ID: 227, 471, 686

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Wrong type used in sizeof

`sizeof` argument does not match pointed type

Description

This defect occurs when both of the following conditions hold:

- You assign the address of a block of memory to a pointer, or transfer data between two blocks of memory. The assignment or copy uses the `sizeof` operator.

  For instance, you initialize a pointer using `malloc(sizeof(`*`type`*`))` or copy data between two addresses using `memcpy(`*`destination_ptr`*`, `*`source_ptr`*`, sizeof(`*`type`*`))`.
- You use an incorrect type as argument of the `sizeof` operator. You use the pointer type instead of the type that the pointer points to.

  For instance, to initialize a *`type`*`*` pointer, you use `malloc(sizeof(`*`type`*`*`))` instead of `malloc(sizeof(`*`type`*`))`.

Risk

Irrespective of what *`type`* stands for, the expression `sizeof(`*`type`*`*`)` always returns a fixed size. The size returned is the pointer size on your platform in bytes. The appearance of `sizeof(`*`type`*`*`)` often indicates an unintended usage. The error can cause allocation of a memory block that is much smaller than what you need and lead to weaknesses such as buffer overflows.

For instance, assume that `structType` is a structure with ten `int` variables. If you initialize a `structType*` pointer using `malloc(sizeof(structType*))` on a 32-bit platform, the pointer is assigned a memory block of four bytes. However, to be allocated completely for one `structType` variable, the `structType*` pointer must point to a memory block of `sizeof(structType) = 10 * sizeof(int)` bytes. The required size is much greater than the actual allocated size of four bytes.

Fix

To initialize a *`type`*`*` pointer, replace `sizeof(`*`type`*`*`)` in your pointer initialization expression with `sizeof(`*`type`*`)`.

Examples

Allocate a Char Array With `sizeof`

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char*) * 5);
    free(str);

}
```

In this example, memory is allocated for the character pointer `str` using a `malloc` of five char pointers. However, `str` is a pointer to a character, not a pointer to a character pointer. Therefore the `sizeof` argument, `char*`, is incorrect.

Correction — Match Pointer Type to `sizeof` Argument

One possible correction is to match the argument to the pointer type. In this example, `str` is a character pointer, therefore the argument must also be a character.

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char) * 5);
    free(str);

}
```

Check Information

Group: Programming
Language: C | C++
Default: On for handwritten code, off for generated code
Command-Line Syntax: `PTR_SIZEOF_MISMATCH`
Impact: High
CWE ID: 467

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Data Flow Defects

Code deactivated by constant false condition

Code segment deactivated by `#if 0` directive or `if(0)` condition

Description

This defect occurs when a block of code is deactivated using a `#if 0` directive or `if(0)` condition.

Risk

A `#if 0` directive or `if(0)` condition is used to temporarily deactivate segments of code. If your production code contains these directives, it means that the deactivation has not been lifted before shipping the code.

Fix

If the segment of code is present for debugging purposes only, remove the segment from production code. If the deactivation occurred by accident, remove the `#if 0` and `#endif` statements.

Often, a segment of code is deactivated for specific conditions, for instance, a specific operating system. Use macros with the `#if` directive to indicate these conditions instead of deactivating the code completely with a `#if 0` directive. For instance, GCC provides macros to detect the Windows operating system:

```
#ifdef _WIN32
   //Code deactivated for all operating systems
   //Other than 32-bit Windows
#endif
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Code Deactivated by Constant False Condition Error

```
#include<stdio.h>
int Trim_Value(int* Arr,int Size,int Cutoff)
{
    int Count=0;

    for(int i=0;i < Size;i++){
        if(Arr[i]>Cutoff){
            Arr[i]=Cutoff;
            Count++;
        }
    }

    #if 0
    /* Defect: Code Segment Deactivated */

    if(Count==0){
        printf("Values less than cutoff.");
```

```
    }
     #endif

    return Count;
}
```

In the preceding code, the `printf` statement is placed within a `#if #endif` directive. The software treats the portion within the directive as code comments and not compiled.

Correction — Change `#if 0` to `#if 1`

Unless you intended to deactivate the `printf` statement, one possible correction is to reactivate the block of code in the `#if #endif` directive. To reactivate the block, change `#if 0` to `#if 1`.

```
#include<stdio.h>
int Trim_Value(int* Arr,int Size,int Cutoff)
{
 int Count=0;

 for(int i=0;i < Size;i++)
     {
      if(Arr[i]>Cutoff)
            {
             Arr[i]=Cutoff;
             Count++;
            }
     }


 /* Fix: Replace #if 0 by #if 1 */
 #if 1
      if(Count==0)
           {
            printf("Values less than cutoff.");
           }
 #endif

 return Count;
}
```

Check Information

Group: Data flow
Language: C | C++
Default: off
Command-Line Syntax: `DEACTIVATED_CODE`
Impact: Low

See Also

`Dead code` | `Find defects (-checkers)` | `Unreachable code` | `Useless if`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Dead code

Code does not execute

Description

This defect occurs when a block of code cannot be reached because of a condition that is always true or false. This defect excludes:

- `Code deactivated by constant false condition`, which checks for directives with compile-time constants such as `#if 0` or `if(0)`.
- `Unreachable code`, which checks for code after a control escape such as `goto`, `break`, or `return`.
- `Useless if`, which checks for if statements that are always true.

Risk

Dead code wastes development time, memory and execution cycles. Developers have to maintain code that is not being executed. Instructions that are not executed still have to be stored and cached.

Dead code often represents legacy code that is no longer used. Cleaning up dead code periodically reduces future maintenance.

Fix

The fix depends on the root cause of the defect. For instance, the root cause can be an error condition that is checked twice on the same execution path, making the second check redundant and the corresponding block dead code.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you see dead code from use of functions such as `isinf` and `isnan`, enable an analysis mode that takes into account non-finite values. See `Consider non finite floats (-allow-non-finite-floats)`.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Dead Code from `if`-Statement

```
#include <stdio.h>

int Return_From_Table(int ch){
```

```
    int table[5];

    /* Create a table */
    for(int i=0;i<=4;i++){
        table[i]=i^2+i+1;
    }

    if(table[ch]>100){ /* Defect: Condition always false */
         return 0;
    }
    return table[ch];
}
```

The maximum value in the array `table` is `4^2+4+1=21`, so the test expression `table[ch]>100` always evaluates to false. The `return 0` in the `if` statement is not executed.

Correction — Remove Dead Code

One possible correction is to remove the `if` condition from the code.

```
#include <stdio.h>

int Return_From_Table(int ch){

    int table[5];

    /* Create a table */
    for(int i=0;i<=4;i++){
        table[i]=i^2+i+1;
    }

    return table[ch];
}
```

Dead Code for `if` with Enumerated Type

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS))
        card = UNKNOWN_SUIT;

    if (card > 7) {
        do_something(card);
    }
}
```

The type `suit` is enumerated with five options. However, the conditional expression `card > 7` always evaluates to false because `card` can be at most 5. The content in the `if` statement is not executed.

Correction — Change Condition

One possible correction is to change the if-condition in the code. In this correction, the 7 is changed to HEART to relate directly to the type of `card`.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS))
        card = UNKNOWN_SUIT;

    if (card > HEARTS) {
        do_something(card);
    }
}
```

Check Information

Group: Data flow
Language: C | C++
Default: On
Command-Line Syntax: DEAD_CODE
Impact: Low
CWE ID: 561

See Also

Code deactivated by constant false condition | Find defects (-checkers) | Unreachable code | Useless if

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Missing return statement

Function does not return value though return type is not `void`

Description

This defect occurs when a function does not return a value along at least one execution path. If the return type of the function is `void`, this error does not occur.

Risk

If a function has a non-`void` return value in its signature, it is expected to return a value. The return value of this function can be used in later computations. If the execution of the function body goes through a path where a `return` statement is missing, the function return value is indeterminate. Computations with this return value can lead to unpredictable results.

Fix

In most cases, you can fix this defect by placing the `return` statement at the end of the function body.

Alternatively, you can identify which execution paths through the function body do not have a `return` statement and add a `return` statement on those paths. Often the result details show a sequence of events that indicate this execution path. You can add a `return` statement at an appropriate point in the path. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

If the analysis flags a missing `return` statement on a path where a process termination function exists, you can make the analysis aware of the process termination function using the option `-termination-functions`.

Examples

Missing or invalid return statement error

```
int AddSquares(int n)
 {
   int i=0;
   int sum=0;

   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
```

```
     return(sum);
    }
 }
/* Defect: No return value if n is not 0*/
```

If `n` is equal to 0, the code does not enter the `if` statement. Therefore, the function `AddSquares` does not return a value if `n` is 0.

Correction — Place Return Statement on Every Execution Path

One possible correction is to return a value in every branch of the `if...else` statement.

```
int AddSquares(int n)
{
  int i=0;
  int sum=0;

  if(n!=0)
   {
    for(i=1;i<=n;i++)
       {
        sum+=i^2;
       }
    return(sum);
   }

  /*Fix: Place a return statement on branches of if-else */
  else
    return 0;
 }
```

Check Information

Group: Data flow
Language: C | C++
Default: On
Command-Line Syntax: `MISSING_RETURN`
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Non-initialized pointer

Pointer not initialized before dereference

Description

This defect occurs when a pointer is not assigned an address before dereference.

Risk

Unless a pointer is explicitly assigned an address, it points to an unpredictable location.

Fix

The fix depends on the root cause of the defect. For instance, you assigned an address to the pointer but the assignment is unreachable.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below. It is a good practice to initialize a pointer to NULL when declaring the pointer.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Non-initialized pointer error

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
      {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
      }

    *pi = j;
    /* Defect: Writing to uninitialized pointer */

    return pi;
}
```

If `prev` is not `NULL`, the pointer `pi` is not assigned an address. However, `pi` is dereferenced on every execution paths, irrespective of whether `prev` is `NULL` or not.

Correction — Initialize Pointer on Every Execution Path

One possible correction is to assign an address to `pi` when `prev` is not `NULL`.

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
       {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
       }
    /* Fix: Initialize pi in branches of if statement  */
    else
        pi = prev;


    *pi = j;

    return pi;
}
```

Check Information

Group: Data flow
Language: C | C++
Default: On
Command-Line Syntax: `NON_INIT_PTR`
Impact: High
CWE ID: 456, 457, 824, 908

See Also

`Find defects (-checkers)` | `Non-initialized variable`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Non-initialized variable

Variable not initialized before use

Description

This defect occurs when a variable is not initialized before its value is read.

Risk

Unless a variable is explicitly initialized, the variable value is unpredictable. You cannot rely on the variable having a specific value.

Fix

The fix depends on the root cause of the defect. For instance, you assigned a value to the variable but the assignment is unreachable or you assigned a value to the variable in one of two branches of a conditional statement. Fix the unreachable code or missing assignment.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below. It is a good practice to initialize a variable at declaration.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Non-initialized variable error

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    int val;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
    /* Defect: val does not have a value if command is not 2 */
}
```

If `command` is not 2, the variable `val` is unassigned. In this case, the return value of function `get_sensor_value` is undetermined.

Correction — Initialize During Declaration

One possible correction is to initialize `val` during declaration so that the initialization is not bypassed on some execution paths.

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    /* Fix: Initialize val */
    int val=0;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
 }
```

`val` is assigned an initial value of 0. When `command` is not equal to 2, the function `get_sensor_value` returns this value.

Check Information

Group: Data flow
Language: C | C++
Default: On
Command-Line Syntax: `NON_INIT_VAR`
Impact: High
CWE ID: 456, 457, 908

See Also

`Find defects (-checkers)` | `Non-initialized pointer`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Partially accessed array

Array partly read or written before end of scope

Description

This defect occurs when an array is partially read or written before the end of array scope. For arrays local to a function, the end of scope occurs when the function ends.

Risk

A partially accessed array often indicates an omission in coding. For instance, when sorting an array using a loop, you used a number of loop iterations such that one array element is never read. The implementation can result in an array that is not fully sorted.

Fix

The fix depends on the root cause of the defect. For instance, if the root cause is a loop with an incorrect number of iterations, change the loop bound or add a step after the loop to access the unread or unwritten elements.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Partially accessed array error

```
int Calc_Sum(void)
{
  int tab[5]={0,1,2,3,4},sum=0;
  /* Defect: tab[4] is not read */

  for (int i=0; i<4;i++) sum+=tab[i];

  return(sum);

 }
```

The array `tab` is only partially read before end of function `Calc_Sum`. While calculating `sum`, `tab[4]` is not included.

Correction — Access Every Array Element

One possible correction is to read every element in the array `tab`.

```
int Calc_Sum(void)
{
  int tab[5]={0,1,2,3,4},sum=0;
```

```
  /* Fix: Include tab[4] in calculating sum */
  for (int i=0; i<5;i++) sum+=tab[i];

  return(sum);

 }
```

Check Information

Group: Data flow
Language: C | C++
Default: On for handwritten code, off for generated code
Command-Line Syntax: `PARTIALLY_ACCESSED_ARRAY`
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Pointer to non-initialized value converted to const pointer

Pointer to constant assigned address that does not contain a value

Description

This defect occurs when a pointer to a constant (`const int*`, `const char*`, etc.) is assigned an address that does not yet contain a value.

Risk

A pointer to a constant stores a value that must not be changed later in the program. If you assign the address of a non-initialized variable to the pointer, it now points to an address with garbage values for the remainder of the program.

Fix

Initialize a variable before assigning its address to a pointer to a constant.

Examples

Pointer to non initialized value converted to const pointer error

```
#include<stdio.h>

void Display_Parity()
 {
  int num,parity;
  const int* num_ptr = &num;
  /* Defect: Address &num does not store a value */

  printf("Enter a number\n:");
  scanf("%d",&num);

  parity=((*num_ptr)%2);
  if(parity==0)
    printf("The number is even.");
  else
    printf("The number is odd.");

 }
```

`num_ptr` is declared as a pointer to a constant. However the variable `num` does not contain a value when `num_ptr` is assigned the address `&num`.

Correction — Store Value in Address Before Assignment to Pointer

One possible correction is to obtain the value of `num` from the user before `&num` is assigned to `num_ptr`.

```
#include<stdio.h>
```

```
void Display_Parity()
 {
  int num,parity;
  const int* num_ptr;

  printf("Enter a number\n:");
  scanf("%d",&num);

 /* Fix: Assign &num to pointer after it receives a value */
  num_ptr=&num;
  parity=((*num_ptr)%2);
  if(parity==0)
    printf("The number is even.");
  else
    printf("The number is odd.");
 }
```

The `scanf` statement stores a value in `&num`. Once the value is stored, it is legitimate to assign `&num` to `num_ptr`.

Check Information

Group: Data flow
Language: C | C++
Default: Off
Command-Line Syntax: `NON_INIT_PTR_CONV`
Impact: Medium

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Static uncalled function

Function with static scope not called in file

Description

This defect occurs when a `static` function is not called in the same file where it is defined.

Risk

Uncalled functions often result from legacy code and cause unnecessary maintenance.

Fix

If the function is not meant to be called, remove the function. If the function is meant for debugging purposes only, wrap the function definition in a debug macro.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Examples

Uncalled function error

Save the following code in the file `Initialize_Value.c`

```
#include <stdlib.h>
#include <stdio.h>

static int Initialize(void)
/* Defect: Function not called */
  {
   int input;
   printf("Enter an integer:");
   scanf("%d",&input);
   return(input);
  }

 void main()
  {
   int num;

   num=0;

   printf("The value of num is %d",num);
  }
```

The `static` function `Initialize` is not called in the file `Initialize_Value.c`.

Correction — Call Function at Least Once

One possible correction is to call `Initialize` at least once in the file `Initialize_Value.c`.

```
#include <stdlib.h>
#include <stdio.h>

static int Initialize(void)
  {
   int input;
   printf("Enter an integer:");
   scanf("%d",&input);
   return(input);
  }

 void main()
  {
   int num;

   /* Fix: Call static function Initialize */
   num=Initialize();

   printf("The value of num is %d",num);
  }
```

Check Information

Group: Data flow
Language: C | C++
Default: Off
Command-Line Syntax: `UNCALLED_FUNC`
Impact: Low
CWE ID: 561

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Unreachable code

Code not executed because of preceding control-flow statements

Description

This defect occurs when a section of code cannot be reached because of a previous break in control flow.

Statements such as `break`, `goto`, and `return`, move the flow of the program to another section or function. Because of this flow escape, the statements following the control-flow code, statistically, do not execute, and therefore the statements are unreachable.

This check also finds code following trivial infinite loops, such as `while(1)`. These types of loops only release the flow of the program by exiting the program. This type of exit causes code after the infinite loop to be unreachable.

Risk

Unreachable code wastes development time, memory and execution cycles. Developers have to maintain code that is not being executed. Instructions that are not executed still have to be stored and cached.

Fix

The fix depends on the intended functionality of the unreachable code. If you want the code to be executed, check the placement of the code or the prior statement that diverts the control flow. For instance, if the unreachable code follows a `return` statement, you might have to switch their order or remove the `return` statement altogether.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Unreachable Code After Return

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void guess(suit s);

suit deal(void){
    suit card = nextcard();
    if( (card < SPADES) || (card > CLUBS) )
        card = UNKNOWN_SUIT;
        return card;

    if (card < HEARTS) {
        guess(card);
    }
    return card;
}
```

In this example, there are missing braces and misleading indentation. The first `return` statement changes the flow of code back to where the function was called. Because of this return statement, the `if`-block and second `return` statement do not execute.

If you correct the indentation and the braces, the error becomes clearer.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void guess(suit s);

suit deal(void){
    suit card = nextcard();
    if( (card < SPADES) || (card > CLUBS) ){
        card = UNKNOWN_SUIT;
    }
    return card;

    if (card < HEARTS) {
        guess(card);
    }
    return card;
}
```

Correction — Remove Return

One possible correction is to remove the escape statement. In this example, remove the first `return` statement to reach the final `if` statement.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void guess(suit s);

suit deal(void){
    suit card = nextcard();
    if( (card < SPADES) || (card > CLUBS) )
    {
        card = UNKNOWN_SUIT;
    }

    if(card < HEARTS)
    {
        guess(card);
    }
    return card;
}
```

Correction — Remove Unreachable Code

Another possible correction is to remove the unreachable code if you do not need it. Because the function does not reach the second `if`-statement, removing it simplifies the code and does not change the program behavior.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void guess(suit s);

suit deal(void){
    suit card = nextcard();
```

```
    if( (card < SPADES) || (card > CLUBS) )
    {
        card = UNKNOWN_SUIT;
    }
    return card;
}
```

Infinite Loop Causing Unreachable Code

```
int add_apples(int apple) {
    int count = 1;
    while(1) {
        if(apple < 99){
            apple++;
            count++;
        }else{
            count--;
        }
    }
    return count;
}
```

In this example, the `while(1)` statement creates an infinite loop. The `return count` statement following this infinite loop is unreachable because the only way to exit this infinite loop is to exit the program.

Correction — Rewrite Loop Condition

One possible correction is to change the loop condition to make the `while` loop finite. In the example correction here, the loop uses the statement from the `if` condition: `apple < 99`.

```
int add_apples1(int apple) {
    int count = 0;
    while(apple < 99) {
        apple++;
        count++;
    }
    if(count == 0)
        count = -1;
    return count;
}
```

Correction — Add a Break Statement

Another possible correction is to add a break from the infinite loop, so there is a possibility of reaching code after the infinite loop. In this example, a `break` is added to the `else` block making the `return count` statement reachable.

```
int add_apples(int apple) {
    int count = 1;
    while(1) {
        if(apple < 99)
        {
            apple++;
            count++;
        }else{
            count--;
            break;
```

```
        }
    }
    return count;
}
```

Correction — Remove Unreachable Code

Another possible correction is to remove the unreachable code. This correction cleans up the code and makes it easier to review and maintain. In this example, remove the return statement and change the function return type to `void`.

```
void add_apples(int apple) {
    int count = 1;
    while(1) {
        if(apple < 99)
        {
            apple++;
            count++;
        }else{
            count--;
        }
    }
}
```

Check Information

Group: Data flow
Language: C | C++
Default: On
Command-Line Syntax: `UNREACHABLE`
Impact: Medium
CWE ID: 561

See Also

`Code deactivated by constant false condition` | `Dead code` | `Find defects (-checkers)` | `Useless if`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Useless if

Unnecessary if conditional

Description

This defect occurs on `if`-statements where the condition is always true. This defect occurs only on if-statements that do not have an else-statement.

This defect shows unnecessary `if`-statements when there is no difference in code execution if the `if`-statement is removed.

Risk

Unnecessary `if` statements often indicate a coding error. Perhaps the `if` condition is coded incorrectly or the `if` statement is not required at all.

Fix

The fix depends on the root cause of the defect. For instance, the root cause can be an error condition that is checked twice on the same execution path, making the second check redundant.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If the redundant condition represents defensive coding practices and you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

`if` with Enumerated Type

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    if (card < 7) {
        do_something(card);
    }
}
```

The type `suit` is enumerated with five options. However, the conditional expression `card < 7` always evaluates to true because `card` can be at most 5. The `if` statement is unnecessary.

Correction — Change Condition

One possible correction is to change the if-condition in the code. In this correction, the 7 is changed to `UNKNOWN_SUIT` to relate directly to the type of `card`.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    if (card > UNKNOWN_SUIT) {
        do_something(card);
    }
}
```

Correction — Remove If

Another possible correction is to remove the if-condition in the code. Because the condition is always true, you can remove the condition to simplify your code.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    do_something(card);
}
```

Check Information

Group: Data flow
Language: C | C++
Default: On
Command-Line Syntax: `USELESS_IF`
Impact: Medium

See Also

`Code deactivated by constant false condition` | `Dead code` | `Find defects (-checkers)` | `Unreachable code`

Topics

“Interpret Polyspace Bug Finder Results”

“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Variable shadowing

Variable hides another variable of same name with nested scope

Description

This defect occurs when a variable hides another variable of the same name in an outer scope.

For instance, if a local variable has the same name as a global variable, the local variable hides the global variable during its lifetime.

Risk

When two variables with the same name exist in an inner and outer scope, any reference to the variable name uses the variable in the inner scope. However, a developer or reviewer might incorrectly expect that the variable in the outer scope was used.

Fix

The fix depends on the root cause of the defect. For instance, suppose you refactor a function such that you use a local static variable in place of a global variable. In this case, the global variable is redundant and you can remove its declaration. Alternatively, if you are not sure if the global variable is used elsewhere, you can modify the name of the local static variable and all references within the function.

If the shadowing is intended and you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Variable Shadowing Error

```
#include <stdio.h>

int fact[5]={1,2,6,24,120};

int factorial(int n)
 {
  int fact=1;
  /*Defect: Local variable hides global array with same name */

  for(int i=1;i<=n;i++)
    fact*=i;

  return(fact);
 }
```

Inside the `factorial` function, the integer variable `fact` hides the global integer array `fact`.

Correction — Change Variable Name

One possible correction is to change the name of one of the variables, preferably the one with more local scope.

```
#include <stdio.h>

int fact[5]={1,2,6,24,120};

int factorial(int n)
 {
  /* Fix: Change name of local variable */
  int f=1;

  for(int i=1;i<=n;i++)
    f*=i;

  return(f);
 }
```

Check Information

Group: Data flow
Language: C | C++
Default: On for handwritten code, off for generated code
Command-Line Syntax: `VAR_SHADOWING`
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Write without a further read

Variable never read after assignment

Description

This defect occurs when a value assigned to a variable is never read.

For instance, you write a value to a variable and then write a second value before reading the previous value. The first write operation is redundant.

Risk

Redundant write operations often indicate programming errors. For instance, you forgot to read the variable between two successive write operations or unintentionally read a different variable.

Fix

Identify the reason why you write to the variable but do not read it later. Look for common programming errors such as accidentally reading a different variable with a similar name.

If you determine that the write operation is redundant, remove the operation.

Examples

Write Without Further Read Error

```
void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();
    /* Defect: Useless write */
}
```

After the variable `level` gets assigned the value `4 * getsensor()`, it is not read.

Correction — Use Value After Assignment

One possible correction is to use the variable `level` after the assignment.

```
#include <stdio.h>

void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();

    /* Fix: Use level after assignment */
    printf("The value is %d", level);
```

```
}
```

The variable `level` is printed, reading the new value.

Check Information

Group: Data flow
Language: C | C++
Default: On for handwritten code, off for generated code
Command-Line Syntax: `USELESS_WRITE`
Impact: Low
CWE ID: 398

See Also

`Find defects (-checkers)|MISRA C:2012 Rule 2.2`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Security Defects

Bad order of dropping privileges

Dropped higher elevated privileges before dropping lower elevated privileges

Description

This defect occurs when you use functions such as `setuid` and `setgid` in the incorrect order, dropping higher elevated privileges before dropping lower elevated privileges. For example, you drop elevated primary group privileges before dropping elevated ancillary group privileges.

Risk

If you drop privileges in the wrong order, you can potentially drop higher privileges that you need to drop lower privileges. The incorrect order can mean that privileges are not dropped compromising the security of your program.

Fix

Respect this order of dropping elevated privileges:

- Drop (elevated) ancillary group privileges, then drop (elevated) primary group privileges.
- Drop (elevated) primary group privileges, then drop (elevated) user privileges.

Examples

Dropping User Privileges First

```
#define _BSD_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>
#define fatal_error() abort()

static void sanitize_privilege_drop_check(uid_t olduid, gid_t oldgid)
{
    if (seteuid(olduid) != -1)
    {
        /* Privileges can be restored, handle error */
        fatal_error();
    }
    if (setegid(oldgid) != -1)
    {
        /* Privileges can be restored, handle error */
        fatal_error();
    }
}
void badprivilegedroporder(void) {
    uid_t
        newuid = getuid(),
        olduid = geteuid();
    gid_t
        newgid = getgid(),
```

```
        oldgid = getegid();

    if (setuid(newuid) == -1) {
        /* handle error condition */
        fatal_error();
    }
    if (setgid(newgid) == -1)  {
        /* handle error condition */
        fatal_error();
    }
    if (olduid == 0) {
        /* drop ancillary groups IDs only possible for root */
        if (setgroups(1, &newgid) == -1) {
            /* handle error condition */
            fatal_error();
        }
    }

    sanitize_privilege_drop_check(olduid, oldgid);
}
```

In this example, there are two privilege drops made in the incorrect order. `setgid` attempts to drop group privileges. However, `setgid` requires the user privileges, which were dropped previously using `setuid`, to perform this function. After dropping group privileges, this function attempts to drop ancillary groups privileges by using `setgroups`. This task requires the higher primary group privileges that were dropped with `setgid`. At the end of this function, it is possible to regain group privileges because the order of dropping privileges was incorrect.

Correction — Reverse Privilege Drop Order

One possible correction is to drop the lowest level privileges first. In this correction, ancillary group privileges are dropped, then primary group privileges are dropped, and finally user privileges are dropped.

```
#define _BSD_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>
#define fatal_error() abort()

static void sanitize_privilege_drop_check(uid_t olduid, gid_t oldgid)
{
    if (seteuid(olduid) != -1)
    {
        /* Privileges can be restored, handle error */
        fatal_error();
    }
    if (setegid(oldgid) != -1)
    {
        /* Privileges can be restored, handle error */
        fatal_error();
    }
}
void badprivilegedroporder(void) {
    uid_t
        newuid = getuid(),
        olduid = geteuid();
```

```
    gid_t
        newgid = getgid(),
        oldgid = getegid();

    if (olduid == 0) {
        /* drop ancillary groups IDs only possible for root */
        if (setgroups(1, &newgid) == -1) {
            /* handle error condition */
            fatal_error();
        }
    }
    if (setgid(getgid()) == -1)  {
        /* handle error condition */
        fatal_error();
    }
    if (setuid(getuid()) == -1) {
        /* handle error condition */
        fatal_error();
    }

    sanitize_privilege_drop_check(olduid, oldgid);
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `BAD_PRIVILEGE_DROP_ORDER`
Impact: High
CWE ID: 250, 696

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2016b

Deterministic random output from constant seed

Seeding routine uses a constant seed making the output deterministic

Description

This defect occurs when you use random standard functions that have deterministic output given a constant seed.

Risk

When some random functions, such as `srand`, `srandom`, and `initstate`, have constant seeds, the results produce the same output every time that your program is run. A hacker can disrupt your program if they know how your program behaves.

Fix

Use a different random standard function or use a nonconstant seed.

Some standard random routines are inherently cryptographically weak on page 3-405, and should not be used for security purposes.

Examples

Random Number Generator Initialization

```
#include <stdlib.h>

void random_num(void)
{
    srand(12345U);
    /* ... */
}
```

This example initializes a random number generator using `srand` with a constant seed. The random number generation is deterministic, making this function cryptographically weak.

Correction — Use Different Random Number Generator

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S
#include <stdlib.h>
#include <stdio.h>

unsigned int random_num_time(void)
{

    unsigned int number;
    errno_t err;
    err = rand_s(&number);
```

```
    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: RAND_SEED_CONSTANT
Impact: Medium
CWE ID: 330, 336

See Also

`Predictable random output from predictable seed` | `Unsafe standard encryption function` | `Vulnerable pseudo-random number generator` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Errno not checked

`errno` is not checked for error conditions following function call

Description

This defect occurs when you call a function that sets `errno` to indicate error conditions, but do not check `errno` after the call. For these functions, checking `errno` is the only reliable way to determine if an error occurred.

Functions that set `errno` on errors include:

- `fgetwc`, `strtol`, and `wcstol`.

  For a comprehensive list of functions, see documentation about errno.
- POSIX `errno`-setting functions such as `encrypt` and `setkey`.

Risk

To see if the function call completed without errors, check `errno` for error values.

The return values of these `errno`-setting functions do not indicate errors. The return value can be one of the following:

- `void`
- Even if an error occurs, the return value can be the same as the value from a successful call. Such return values are called in-band error indicators.

You can determine if an error occurred only by checking `errno`.

For instance, `strtol` converts a string to a long integer and returns the integer. If the result of conversion overflows, the function returns `LONG_MAX` and sets `errno` to `ERANGE`. However, the function can also return `LONG_MAX` from a successful conversion. Only by checking `errno` can you distinguish between an error and a successful conversion.

Fix

Before calling the function, set `errno` to zero.

After the function call, to see if an error occurred, compare `errno` to zero. Alternatively, compare `errno` to known error indicator values. For instance, `strtol` sets `errno` to `ERANGE` to indicate errors.

The error message in the Polyspace result shows the error indicator value that you can compare to.

Examples

`errno` Not Checked After Call to `strtol`

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
```

```
int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    long val = strtol(str, &endptr, base);
    printf("Return value of strtol() = %ld\n", val);
}
```

You are using the return value of `strtol` without checking `errno`.

Correction — Check `errno` After Call

Before calling `strtol`, set `errno` to zero . After a call to `strtol`, check the return value for `LONG_MIN` or `LONG_MAX` and `errno` for `ERANGE`.

```
#include<stdlib.h>
#include<stdio.h>
#include<errno.h>
#include<limits.h>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    errno = 0;
    long val = strtol(str, &endptr, base);
    if((val == LONG_MIN || val == LONG_MAX) && errno == ERANGE) {
         printf("strtol error");
         exit(EXIT_FAILURE);
    }
    printf("Return value of strtol() = %ld\n", val);
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `ERRNO_NOT_CHECKED`
Impact: Medium
CWE ID: 253, 391

See Also

`Errno not reset` | `Find defects (-checkers)` | `Misuse of errno` | `Returned value of a sensitive function not checked`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Execution of a binary from a relative path can be controlled by an external actor

Command with relative path is vulnerable to malicious attack

Description

This defect occurs when you call an external command with a relative path or without a path.

This defect also finds results that the **Execution of externally controlled command** defect checker finds.

Risk

By using a relative path or no path to call an external command, your program uses an unsafe search process to find the command. An attacker can control the search process and replace the intended command with a command of their own.

Fix

When you call an external command, specify the full path.

Examples

Call Command with Relative Path

```
# define _GNU_SOURCE
# include <sys/types.h>
# include <sys/socket.h>
# include <unistd.h>
# include <stdio.h>
# include <stdlib.h>
# include <wchar.h>
# include <string.h>
# define MAX_BUFFER 100

void rel_path()
{
    char * data;
    char data_buf[MAX_BUFFER] = "";
    data = data_buf;

    strcpy(data, "ls -la");
    FILE *pipe;
    pipe = popen(data, "wb");
    if (pipe != NULL) pclose(pipe);
}
```

In this example, Bug Finder flags `popen` because it tries to call `ls -la` using a relative path to the `ls` command. An attacker can manipulate the command to use a malicious version.

Correction — Use Full Path

One possible correction is to use the full path when calling the command.

```
# define _GNU_SOURCE
# include <sys/types.h>
# include <sys/socket.h>
# include <unistd.h>
# include <stdio.h>
# include <stdlib.h>
# include <wchar.h>
# include <string.h>
# define MAX_BUFFER 100

void rel_path()
{
    char * data;
    char data_buf[MAX_BUFFER] = "";
    data = data_buf;

    strcpy(data, "/usr/bin/ls -la");
    FILE *pipe;
    pipe = popen(data, "wb");
    if (pipe != NULL) pclose(pipe);
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `RELATIVE_PATH_CMD`
Impact: Medium
CWE ID: 114, 427

See Also

`Load of library from a relative path can be controlled by an external actor` | `Vulnerable path manipulation` | `Execution of externally controlled command` | `Command executed from externally controlled path` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

File access between time of check and use (TOCTOU)

File or folder might change state due to access race

Description

This defect occurs when a race condition happens between checking the existence of a file or folder, and using the file or folder.

Risk

An attacker can access and manipulate your file between your check for the file and your use of a file. Symbolic links are particularly risky because an attacker can change where your symbolic link points.

Fix

Before using a file, do not check its status. Instead, use the file and check the results afterward.

Examples

Check File Before Using

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern void print_tofile(FILE* f);

void toctou(char * log_path) {
    if (access(log_path, W_OK)==0) {
        FILE* f = fopen(log_path, "w");
        if (f) {
            print_tofile(f);
            fclose(f);
        }
    }
}
```

In this example, before opening and using the file, the function checks if the file exists. However, an attacker can change the file between the first and second lines of the function.

Correction — Open Then Check

One possible correction is to open the file, and then check the existence and contents afterward.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern void print_tofile(FILE* f);

void toctou(char * log_path) {
```

```
    int fd = open(log_path, O_WRONLY);
    if (fd!=-1) {
        FILE *f = fdopen(fd, "w");
        if (f) {
            print_tofile(f);
            fclose(f);
        }
    }
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: TOCTOU
Impact: Medium
CWE ID: 367

See Also

`Data race` | `Bad file access mode or status` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

File descriptor exposure to child process

Copied file descriptor used in multiple processes

Description

This defect occurs when a process is forked and the child process uses file descriptors inherited from the parent process.

Risk

When you fork a child process, file descriptors are copied from the parent process, which means that you can have concurrent operations on the same file. Use of the same file descriptor in the parent and child processes can lead to race conditions that may not be caught during standard debugging. If you do not properly manage the file descriptor permissions and privileges, the file content is vulnerable to attacks targeting the child process.

Fix

Check that the file has not been modified before forking the process. Close all inherited file descriptors and reopen them with stricter permissions and privileges, such as read-only permission.

Examples

File Descriptor Accessed from Forked Process

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <unistd.h>
# include <fcntl.h>
# include <sys/types.h>
# include <sys/stat.h>



const char *test_file="/home/user/test.txt";

void func(void)
{
    char c;
    pid_t pid;
    /* create file descriptor in read and write mode */
    int fd = open(test_file, O_RDWR);
    if (fd == -1)
    {
        /* Handle error */
        abort();
    }
    /* fork process */
    pid = fork();
    if (pid == -1)
    {
```

```
        /* Handle error */
        abort();
    }
    else if (pid == 0)
    {   /* Child process accesses file descriptor inherited
        from parent process */
        (void)read(fd, &c, 1);
    }
    else
    {   /* Parent process access same file descriptor as
        child process */
        (void)read(fd, &c, 1);
    }
}
```

In this example, a file descriptor `fd` is created in read and write mode. The process is then forked. The child process inherits and accesses `fd` with the same permissions as the parent process. A race condition exists between the parent and child processes. The contents of the file is vulnerable to attacks through the child process.

Correction — Close and Reopen Inherited File Descriptor

After you create the file descriptor, check the file for tampering. Then, close the inherited file descriptor in the child process and reopen it in read-only mode.

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <unistd.h>
# include <fcntl.h>
# include <sys/types.h>
# include <sys/stat.h>


const char *test_file="/home/user/test.txt";

void func(void)
{
    char c;
    pid_t pid;

    /* Get the state of file for further file tampering checking */

    /* create file descriptor in read and write mode */
    int fd = open(test_file, O_RDWR);
    if (fd == -1)
    {
        /* Handle error */
        abort();
    }

    /* Be sure the file was not tampered with while opening */

    /* fork process */

    pid = fork();
    if (pid == -1)
```

```
    {
        /* Handle error */
        (void)close(fd);
        abort();
    }
    else if (pid == 0)
    {  /* Close file descriptor in child process and repoen
        it in read only mode */

        (void)close(fd);
        fd = open(test_file, O_RDONLY);
        if (fd == -1)
        {
            /* Handle error */
            abort();
        }


        (void)read(fd, &c, 1);
        (void)close(fd);
    }
    else
    {  /* Parent acceses original file descriptor */
        (void)read(fd, &c, 1);
        (void)close(fd);
    }
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `FILE_EXPOSURE_TO_CHILD`
Impact: Medium
CWE ID: 362

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

File manipulation after chroot() without chdir("/")

Path-related vulnerabilities for file manipulated after call to chroot

Description

This defect occurs when you have access to a file system outside of the jail created by chroot. By calling chroot, you create a file system jail that confines access to a specific file subsystem. However, this jail is ineffective if you do not call chdir("/").

Risk

If you do not call chdir("/") after creating a chroot jail, file manipulation functions that takes a path as an argument can access files outside of the jail. An attacker can still manipulate files outside the subsystem that you specified, making the chroot jail ineffective.

Fix

After calling chroot, call chdir("/") to make your chroot jail more secure.

Examples

Open File in chroot-jail

```
#include <unistd.h>
#include <stdio.h>

const char root_path[] = "/var/ftproot";
const char log_path[] = "file.log";
FILE* chrootmisuse() {
    FILE* res;
    chroot(root_path);
    chdir("base");
    res = fopen(log_path, "r");
    return res;
}
```

This example uses chroot to create a chroot-jail. However, to use the chroot jail securely, you must call chdir("\") afterward. This example calls chdir("base"), which is not equivalent. Bug Finder also flags fopen because fopen opens a file in the vulnerable chroot-jail.

Correction — Call chdir("/")

Before opening files, call chdir("/").

```
#include <unistd.h>
#include <stdio.h>

const char root_path[] = "/var/ftproot";
const char log_path[] = "file.log";
FILE* chrootmisuse() {
    FILE* res;
```

```
    chroot(root_path);
    chdir("/");
    res = fopen(log_path, "r");
    return res;
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: CHROOT_MISUSE
Impact: Medium
CWE ID: 243, 922

See Also

`Umask used with chmod-style arguments` | `Vulnerable path manipulation` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Function pointer assigned with absolute address

Constant expression is used as function address is vulnerable to code injection

Description

This defect occurs when a function pointer is assigned an absolute address.

Bug Finder considers expressions with any combination of literal constants as an absolute address. The one exception is when the value of the expression is zero.

Risk

Using a fixed address is not portable because it is possible that the address is invalid on other platforms.

An attacker can inject code at the absolute address, causing your program to execute arbitrary, possibly malicious, code.

Fix

Do not use an absolute address with function pointers.

Examples

Function Pointer Address Assignment

```
extern int func0(int i, char c);
typedef int (*FuncPtr) (int, char);

FuncPtr funcptrabsoluteaddr() {
    return (FuncPtr)0x08040000;
}
```

In this example, the function returns a function pointer to the address `0x08040000`. If an attacker knows this absolute address, an attacker can compromise your program.

Correction — Function Address

One possible correction is to use the address of an existing function instead.

```
extern int func0(int i, char c);
typedef int (*FuncPtr) (int, char);

FuncPtr funcptrabsoluteaddr() {
    return &func0;
}
```

Result Information

Group: Security
Language: C | C++
Default: Off

Command-Line Syntax: `FUNC_PTR_ABSOLUTE_ADDR`
Impact: Low
CWE ID: 587

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Hard-coded sensitive data

Sensitive data is exposed in code, for instance as string literals

Description

This defect occurs when data that is potentially sensitive is directly exposed in the code, for instance, as string literals. The checker identifies certain data as sensitive from their use in certain functions such as password encryption functions.

Following data can be potentially sensitive.

Type of Data	Functions That Indicate Sensitive Nature of Information
Host name	• `sethostname`, `setdomainname`, `gethostbyname`, `gethostbyname2`, `getaddrinfo`, `gethostbyname_r`, `gethostbyname2_r` (string argument) • `inet_aton`, `inet_pton`, `inet_net_pton`, `inet_addr`, `inet_network` (string argument) • `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (2nd argument)
Password	• `CreateProcessWithLogonW`, `LogonUser` (1st argument) • `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (3rd argument)
Database	• MySQL: `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (4th argument) • SQLite: `sqlite3_open`, `sqlite3_open16`, `sqlite3_open_v2` (1st argument) • PostgreSQL: `PQconnectdb` • Microsoft SQL: `SQLDriverConnect` (3rd argument)
User name	• `getpw`, `getpwnam`, `getpwnam_r`, `getpwuid`, `getpwuid_r`
Salt	`crypt`, `crypt_r` (2nd argument)

Type of Data	Functions That Indicate Sensitive Nature of Information
Cryptography keys and initialization vectors	OpenSSL: • `EVP_CipherInit`, `EVP_EncryptInit`, `EVP_DecryptInit` (3rd argument) • `EVP_CipherInit_ex`, `EVP_EncryptInit_ex`, `EVP_DecryptInit_ex` (4th argument)
Seed	• `srand`, `srandom`, `initstate` (1st argument) • OpenSSL: `RAND_seed`, `RAND_add`

Risk

Information that is hardcoded can be queried from binaries generated from the code.

Fix

Avoid hard coding sensitive information.

Examples

Sensitive Data Exposed Through String Literals

```
// Typically, you include the header "mysql.h" with function and type declarations.
// In this example, only the required lines from the header are quoted.

typedef struct _MYSQL MYSQL;

MYSQL *mysql_real_connect(MYSQL *mysql,
                          const char *host, const char *user, const char *passwd,
                          const char *db, unsigned int port, const char *unix_socket,
                          unsigned long client_flag);

typedef void * DbHandle;
extern MYSQL *sql;

// File that uses functions from "mysql.h"
const char *host = "localhost";
char *user = "guest";
char *passwd;

DbHandle connect_to_database_server(const char *db)
{
    passwd = (char*)"guest";
    return (DbHandle)
        mysql_real_connect (sql, host, user, passwd, db, 0, 0x0, 0);
}
```

In this example, the arguments `host` (host name), `user` (user name), and `passwd` (password) are string literals and directly exposed in the code.

Querying the generated binary for ASCII strings can reveal this information.

Correction – Read Sensitive Data from Secured Configuration Files

One possible correction is to read the data from a configuration file. In the following corrected example, the call to function `connect_to_database_server_init` presumably reads the host name, user name, and password into its arguments from a secured configuration file.

```
// Typically, you include the header "mysql.h" with function and type declarations.
// In this example, only the required lines from the header are quoted.

typedef struct _MYSQL MYSQL;

MYSQL *mysql_real_connect(MYSQL *mysql,
                          const char *host, const char *user, const char *passwd,
                          const char *db, unsigned int port, const char *unix_socket,
                          unsigned long client_flag);

typedef void * DbHandle;
extern MYSQL *sql;

// File that uses functions from "mysql.h"

DbHandle connect_to_database_server(const char *db)
{
    const char *host_from_cfg;
    const char *user_from_cfg;
    const char *passwd_from_cfg;
    const char *db_from_cfg;
    if (connect_to_database_server_init(&host_from_cfg,
                                        &user_from_cfg,
                                        &passwd_from_cfg,
                                        &db_from_cfg))
    {
        return (DbHandle)
            mysql_real_connect (sql, host_from_cfg, user_from_cfg,
                        passwd_from_cfg, db_from_cfg,  0, 0x0, 0);
    }
    else
        return (DbHandle)0x0;
}
```

Check Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `HARD_CODED_SENSITIVE_DATA`
Impact: Medium

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Inappropriate I/O operation on device files

Operation can result in security vulnerabilities or a system failure

Description

This defect occurs when you do not check whether a file name parameter refers to a device file before you pass it to these functions:

- `fopen()`
- `fopen_s()`
- `freopen()`
- `remove()`
- `rename()`
- `CreateFile()`
- `CreateFileA()`
- `CreateFileW()`
- `_wfopen()`
- `_wfopen_s()`

Device files are files in a file system that provide an interface to device drivers. You can use these files to interact with devices.

Inappropriate I/O operation on device files does not raise a defect when:

- You use `stat` or `lstat`-family functions to check the file name parameter before calling the previously listed functions.
- You use a string comparison function to compare the file name against a list of device file names.

Risk

Operations appropriate only for regular files but performed on device files can result in denial-of-service attacks, other security vulnerabilities, or system failures.

Fix

Before you perform an I/O operation on a file:

- Use `stat()`, `lstat()`, or an equivalent function to check whether the file name parameter refers to a regular file.
- Use a string comparison function to compare the file name against a list of device file names.

Examples

Using `fopen()` Without Checking `file_name`

```
#include <stdio.h>
#include <string.h>
```

```
#define SIZE1024 1024

FILE* func()
{

    FILE* f;
    const char file_name[SIZE1024] = "./tmp/file";

    if ((f = fopen(file_name, "w")) == NULL) {
        /*handle error */
    };
    /*operate on file */
}
```

In this example, `func()` operates on the file `file_name` without checking whether it is a regular file. If `file_name` is a device file, attempts to access it can result in a system failure.

Correction — Check File with `lstat()` Before Calling `fopen()`

One possible correction is to use `lstat()` and the `S_ISREG` macro to check whether the file is a regular file. This solution contains a `TOCTOU` race condition that can allow an attacker to modify the file after you check it but before the call to fopen(). To prevent this vulnerability, ensure that `file_name` refers to a file in a secure folder.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>

#define SIZE1024 1024

FILE* func()
{

    FILE* f;
    const char file_name[SIZE1024] = "./tmp/file";
    struct stat orig_st;
    if ((lstat(file_name, &orig_st) != 0) ||
        (!S_ISREG(orig_st.st_mode))) {
        exit(0);
    }
    if ((f = fopen(file_name, "w")) == NULL) {
        /*handle error */
    };
    /*operate on file */
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `INAPPROPRIATE_IO_ON_DEVICE`
Impact: Medium
CWE ID: 67

See Also

`File access between time of check and use (TOCTOU)` | `Find defects (-checkers)` | `Opening previously opened resource` | `Resource leak` | `Returned value of a sensitive function not checked` | `Vulnerable path manipulation`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Incorrect order of network connection operations

Socket is not correctly established due to bad order of connection steps or missing steps

Description

This defect occurs when you perform operations on a network connection at the wrong point of the connection lifecycle.

Risk

Sending or receiving data to an incorrectly connected socket can cause unexpected behavior or disclosure of sensitive information.

If you do not connect your socket correctly or change the connection by mistake, you can send sensitive data to an unexpected port. You can also get unexpected data from an incorrect socket.

Fix

During socket connection and communication, check the return of each call and the length of the data.

Before reading, writing, sending, or receiving information, create sockets in this order:

- For a connection-oriented server socket (`SOCK_STREAM` or `SOCK_SEQPACKET`):

  ```
  socket(...);
  bind(...);
  listen(...);
  accept(...);
  ```

- For a connectionless server socket (`SOCK_DGRAM`):

  ```
  socket(...);
  bind(...);
  ```

- For a client socket (connection-oriented or connectionless):

  ```
  socket(...);
  connect(...);
  ```

Examples

Connecting a Connection-Oriented Server Socket

```
# include <stdio.h>
# include <string.h>
# include <time.h>
# include <arpa/inet.h>
# include <unistd.h>

enum { BUF_SIZE=1025 };

volatile int rd;
```

```
int stream_socket_server(int argc, char *argv[])
{
    int listenfd = 0, connfd = 0;
    struct sockaddr_in serv_addr;

    char sendBuff[BUF_SIZE];
    time_t ticks;
    struct tm * timeinfo;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&serv_addr, 48, sizeof(serv_addr));
    memset(sendBuff, 48, sizeof(sendBuff));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(5000);

    bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));

    listen(listenfd, 10);

    while(1)
    {
        connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);

        ticks = time(NULL);
        timeinfo = localtime(&ticks);
        strftime (sendBuff,BUF_SIZE,"%I:%M%p.",timeinfo);

        write(listenfd, sendBuff, strlen(sendBuff));

        close(connfd);
        sleep(1);
    }
}
```

This example creates a connection-oriented network connection. The function calls the correct functions in the correct order: `socket`, `bind`, `listen`, `accept`. However, the program should write to the `connfd` socket instead of the `listenfd` socket.

Correction — Use Safe Socket

One possible correction is to write to the `connfd` function instead of the `listenfd` socket.

```
# include <stdio.h>
# include <string.h>
# include <time.h>
# include <arpa/inet.h>
# include <unistd.h>

enum { BUF_SIZE=1025 };

volatile int rd;

int stream_socket_server_good(int argc, char *argv[])
{
    int listenfd = 0, connfd = 0;
    struct sockaddr_in serv_addr;
```

```
    char sendBuff[BUF_SIZE];
    time_t ticks;
    struct tm * timeinfo;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&serv_addr, 48, sizeof(serv_addr));
    memset(sendBuff, 48, sizeof(sendBuff));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(5000);

    bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    listen(listenfd, 10);

    while(1)
    {
        connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
        ticks = time(NULL);
        timeinfo = localtime(&ticks);
        strftime (sendBuff,BUF_SIZE,"%I:%M%p.",timeinfo);
        write(connfd, sendBuff, strlen(sendBuff));
        close(connfd);
        sleep(1);
    }
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: BAD_NETWORK_CONNECT_ORDER
Impact: Medium
CWE ID: 666

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Information leak via structure padding

Padding bytes can contain sensitive information

Description

This defect occurs when you do not initialize the padding data of a structure or union before passing it across a trust boundary. A compiler adds padding bytes to the structure or union to ensure a proper memory alignment of its members. The bit-fields of the storage units can also have padding bits.

Information leak via structure padding raises a defect when:

- You call an untrusted function with structure or union pointer type argument containing uninitialized padding data.

  All external functions are considered untrusted.
- You copy or assign a structure or union containing uninitialized padding data to an untrusted object.

  All external structure or union objects, the output parameters of all externally linked functions, and the return pointer of all external functions are considered untrusted objects.

Risk

The padding bytes of the passed structure or union might contain sensitive information that an untrusted source can access.

Fix

- Prevent the addition of padding bytes for memory alignment by using the `pack` pragma or attribute supported by your compiler.
- Explicitly declare and initialize padding bytes as fields within the structure or union.
- Explicitly declare and initialize bit-fields corresponding to padding bits, even if you use the `pack` pragma or attribute supported by your compiler.

Examples

Structure with Padding Bytes Passed to External Function

```
#include <stddef.h>
#include <stdlib.h>
#include <string.h>

typedef struct s_padding
{
 /* Padding bytes may be introduced between
 * 'char c' and 'int i'
 */
        char c;
    int i;

/*Padding bits may be introduced around the bit-fields
```

```
* even if you use "#pragma pack" (Windows) or
* __attribute__((__packed__)) (GNU)*/

    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;



/* External function */
extern void copy_object(void *out, void *in, size_t s);

void func(void *out_buffer)
{
/*Padding bytes not initialized*/

    S_Padding s = {'A', 10, 1, 3, {}};
/*Structure passed to external function*/

    copy_object((void *)out_buffer, (void *)&s, sizeof(s));
}

void main(void)
{
    S_Padding s1;
    func(&s1);
}
```

In this example, structure `s1` can have padding bytes between the `char c` and `int i` members. The bit-fields of the storage units of the structure can also contain padding bits. The content of the padding bytes and bits is accessible to an untrusted source when `s1` is passed to `func`.

Correction — Use pack Pragma to Prevent Padding Bytes

One possible correction in Microsoft Visual Studiois to use `#pragma pack()` to prevent padding bytes between the structure members. To prevent padding bits in the bit-fields of `s1`, explicitly declare and initialize the bit-fields even if you use `#pragma pack()`.

```
 #include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#define CHAR_BIT 8

#pragma pack(push, 1)

typedef struct s_padding
{
/*No Padding bytes when you use "#pragma pack" (Windows) or
* __attribute__((__packed__)) (GNU)*/
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
/* Padding bits explicitely declared */
    unsigned int bf_filler : sizeof(unsigned) * CHAR_BIT - 3;
```

```
    unsigned char buffer[20];
}

    S_Padding;

#pragma pack(pop)


/* External function */
extern void copy_object(void *out, void *in, size_t s);



void func(void *out_buffer)
{
    S_Padding s = {'A', 10, 1, 3, 0 /* padding bits */, {}};
    copy_object((void *)out_buffer, (void *)&s, sizeof(s));
}

void main(void)
{
    S_Padding s1;
    func(&s1);
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: PADDING_INFO_LEAK
Impact: Low

See Also

Find defects (-checkers) | Invalid assumptions about memory organization | Large pass-by-value argument | Memory comparison of padding data | Sensitive heap memory not cleared before release | Uncleared sensitive data in stack | Use of memset with size argument zero

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Load of library from a relative path can be controlled by an external actor

Library loaded with relative path is vulnerable to malicious attacks

Description

This defect occurs when library loading routines that load an external library use a relative path or do not use a path at all.

Risk

By using a relative path or no path to load an external library, your program uses an unsafe search process to find the library. An attacker can control the search process and replace the intended library with a library of their own.

Fix

When you load an external library, specify the full path.

Examples

Open Library with Library Name

```
#include <dlfcn.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <stdio.h>

void relative_path()
{
    dlopen("liberty.dll",RTLD_LAZY);
}
```

In this example, `dlopen` opens the `liberty` library by calling only the name of the library. However, this call to the library uses a relative path to find the library, which is unsafe.

Correction — Use Full Path to Library

One possible correction is to use the full path to the library when you load it into your program.

```
#include <dlfcn.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <stdio.h>

void relative_path()
{
    dlopen("/home/my_libs/library/liberty.dll",RTLD_LAZY);
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `RELATIVE_PATH_LIB`
Impact: Medium
CWE ID: 114, 427

See Also

`Execution of a binary from a relative path can be controlled by an external actor` | `Vulnerable path manipulation` | `Library loaded from externally controlled path` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Mismatch between data length and size

Data size argument is not computed from actual data length

Description

This defect occurs when you do not check the length argument and data buffer argument of memory copying functions such as `memcpy`, `memset`, or `memmove`, to protect against buffer overflows.

Risk

If an attacker can manipulate the data buffer or length argument, the attacker can cause buffer overflow by making the actual data size smaller than the length.

This mismatch in length allows the attacker to copy memory past the data buffer to a new location. If the extra memory contains sensitive information, the attacker can now access that data.

This defect is similar to the SSL Heartbleed bug.

Fix

When copying or manipulating memory, compute the length argument directly from the data so that the sizes match.

Examples

Copy Buffer of Data

```
#include <stdlib.h>
#include <string.h>

typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    memcpy(&(beta.data[num]), os->data + 2, length);

    return(1);
}
```

This function copies the buffer `alpha` into a buffer `beta`. However, the `length` variable is not related to `data+2`.

Correction — Check Buffer Length

One possible correction is to check the length of your buffer against the maximum value minus 2. This check ensures that you have enough space to copy the data to the `beta` structure.

```
#include <stdlib.h>
#include <string.h>

typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    if (length<(os->max -2)) {
        memcpy(&(beta.data[num]), os->data + 2, length);
    }

    return(1);

}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `DATA_LENGTH_MISMATCH`
Impact: Medium
CWE ID: 130, 240

See Also

`Copy of overlapping memory` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Missing case for switch condition

`switch` variable not covered by cases and default case is missing

Description

This defect occurs when the `switch` variable can take values that are not covered by a `case` statement.

Note Bug Finder only raises a defect if the switch variable is not full range.

Risk

If the `switch` variable takes a value that is not covered by a `case` statement, your program can have unintended behavior.

A switch-statement that makes a security decision is particularly vulnerable when all possible values are not explicitly handled. An attacker can use this situation to deviate the normal execution flow.

Fix

It is good practice to use a `default` statement as a catch-all for values that are not covered by a `case` statement. Even if the `switch` variable takes an unintended value, the resulting behavior can be anticipated.

Examples

Missing Default Condition

```
#include <stdio.h>
#include <string.h>

typedef enum E
{
    ADMIN=1,
    GUEST,
    UNKNOWN = 0
} LOGIN;

static LOGIN system_access(const char *username) {
  LOGIN user = UNKNOWN;

  if ( strcmp(username, "root") == 0 )
    user = ADMIN;

  if ( strcmp(username, "friend") == 0 )
    user = GUEST;

  return user;
}
```

```
int identify_bad_user(const char * username)
{
    int r=0;

    switch( system_access(username) )
    {
    case ADMIN:
        r = 1;
        break;
    case GUEST:
        r = 2;
    }

    printf("Welcome!\n");
    return r;
}
```

In this example, the `enum` parameter `User` can take a value `UNKNOWN` that is not covered by a `case` statement.

Correction — Add a Default Condition

One possible correction is to add a default condition for possible values that are not covered by a `case` statement.

```
#include <stdio.h>
#include <string.h>

typedef enum E
{
    ADMIN=1,
    GUEST,
    UNKNOWN = 0
} LOGIN;

static LOGIN system_access(const char *username) {
  LOGIN user = UNKNOWN;

  if ( strcmp(username, "root") == 0 )
    user = ADMIN;

  if ( strcmp(username, "friend") == 0 )
    user = GUEST;

  return user;
}

int identify_bad_user(const char * username)
{
    int r=0;

    switch( system_access(username) )
    {
    case ADMIN:
        r = 1;
        break;
    case GUEST:
        r = 2;
```

```
    break;
    default:
        printf("Invalid login credentials!\n");
    }

    printf("Welcome!\n");
    return r;
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `MISSING_SWITCH_CASE`
Impact: Low
CWE ID: 478

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Misuse of readlink()

Third argument of `readlink` does not leave space for null terminator in buffer

Description

This defect occurs when you pass a buffer size argument to `readlink()` that does not leave space for a null terminator in the buffer.

For instance:

```
ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf));
```

The third argument is exactly equal to the size of the second argument. For large enough symbolic links, this use of `readlink()` does not leave space to enter a null terminator.

Risk

The `readlink()` function copies the content of a symbolic link (first argument) to a buffer (second argument). However, the function does not append a null terminator to the copied content. After using `readlink()`, you must explicitly add a null terminator to the buffer.

If you fill the entire buffer when using `readlink`, you do not leave space for this null terminator.

Fix

When using the `readlink()` function, make sure that the third argument is one less than the buffer size.

Then, append a null terminator to the buffer. To determine where to add the null terminator, check the return value of `readlink()`. If the return value is -1, an error has occurred. Otherwise, the return value is the number of characters (bytes) copied.

Examples

Incorrect Size Argument of `readlink`

```
#include <unistd.h>

#define SIZE1024 1024

extern void display_path(const char *);

void func() {
    char buf[SIZE1024];
    ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf));
    if (len > 0) {
        buf[len - 1] = '\0';
    }
    display_path(buf);
}
```

In this example, the third argument of `readlink` is exactly the size of the buffer (second argument). If the first argument is long enough, this use of `readlink` does not leave space for the null terminator.

Also, if no characters are copied, the return value of `readlink` is 0. The following statement leads to a buffer underflow when `len` is 0.

```
buf[len - 1] = '\0';
```

Correction — Make Sure Size Argument is One Less Than Buffer Size

One possible correction is to make sure that the third argument of `readlink` is one less than size of the second argument.

The following corrected code also accounts for `readlink` returning 0.

```
#include <stdlib.h>
#include <unistd.h>

#define fatal_error() abort()
#define SIZE1024 1024

extern void display_path(const char *);

void func() {
    char buf[SIZE1024];
    ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf) - 1);
    if (len != -1) {
        buf[len] = '\0';
        display_path(buf);
    }
    else {
        /* Handle error */
        fatal_error();
    }
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `READLINK_MISUSE`
Impact: Medium
CWE ID: 170

See Also

`Array access out of bounds` | `File access between time of check and use (TOCTOU)` | `Find defects (-checkers)` | `Invalid use of standard library string routine` | `Pointer access out of bounds` | `Returned value of a sensitive function not checked`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Predictable random output from predictable seed

Seeding routine uses a predictable seed making the output predictable

Description

This defect occurs when you use random standard functions with a nonconstant but predictable seed. Examples of predictable seed generators are `time`, `gettimeofday`, and `getpid`.

Risk

When you use predictable seed values for random number generation, your random numbers are also predictable. A hacker can disrupt your program if they know how your program behaves.

Fix

You can use a different function to generate less predictable seeds.

You can also use a different random number generator that does not require a seed. For example, the Windows API function `rand_s` seeds itself by default. It uses information from the entire system, for example, system time, thread ids, system counter, and memory clusters. This information is more random and a user cannot access this information.

Some standard random routines are inherently cryptographically weak on page 3-405, and should not be used for security purposes.

Examples

Seed as an Argument

```
#include <stdlib.h>
#include <time.h>

void seed_rng(int seed)
{
    srand(seed);
}

int generate_num(void)
{
    seed_rng(time(NULL) + 3);
    /* ... */
}
```

This example uses `srand` to start the random number generator with `seed` as the seed. However, `seed` is predictable because the function `time` generates it. So, an attacker can predict the random numbers generated by `srand`.

Correction — Use Different Random Number Generator

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int generate_num(void)
{
    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: RAND_SEED_PREDICTABLE
Impact: Medium
CWE ID: 330, 337

See Also

`Deterministic random output from constant seed` | `Unsafe standard encryption function` | `Vulnerable pseudo-random number generator` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Privilege drop not verified

Attacker can gain unintended elevated access to program

Description

This defect occurs when you relinquish privileges using functions such as `setuid` but do not verify that the privileges were actually dropped before exiting your function.

Risk

If privilege relinquishment fails, an attacker can regain elevated privileges and have more access to your program than intended. This security hole can cause unexpected behavior in your code if left open.

Fix

Before the end of scope, verify that the privileges that you dropped were actually dropped.

Examples

Drop Privileges Within a Function

```
#define _BSD_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>
#define fatal_error() abort()
extern int need_more_privileges;

void missingprivilegedropcheck()
{
    /* Code intended to run with elevated privileges */

    /* Temporarily drop elevated privileges */
    if (seteuid(getuid()) != 0) {
        /* Handle error */
        fatal_error();
    }

    /* Code intended to run with lower privileges */

    if (need_more_privileges) {
        /* Restore elevated privileges */
        if (seteuid(0) != 0) {
            /* Handle error */
            fatal_error();
        }
        /* Code intended to run with elevated privileges */
    }

    /* ... */
```

```
    /* Permanently drop elevated privileges */
    if (setuid(getuid()) != 0) {
        /* Handle error */
        fatal_error();
    }

    /* Code intended to run with lower privileges */
}
```

In this example, privileges are elevated and dropped to run code with the intended privilege level. When privileges are dropped, the privilege level before exiting the function body is not verified. A malicious attacker can regain their elevated privileges.

Correction — Verify Privilege Drop

One possible correction is to use `setuid` to verify that the privileges were dropped.

```
#define _BSD_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>
#define fatal_error() abort()
extern int need_more_privileges;

void missingprivilegedropcheck()
{
    /* Store the privileged ID for later verification */
    uid_t privid = geteuid();

    /* Code intended to run with elevated privileges   */

    /* Temporarily drop elevated privileges */
    if (seteuid(getuid()) != 0) {
        /* Handle error */
        fatal_error();
    }

    /* Code intended to run with lower privileges  */

    if (need_more_privileges) {
        /* Restore elevated Privileges */
        if (seteuid(privid) != 0) {
            /* Handle error */
            fatal_error();
        }
        /* Code intended to run with elevated privileges   */
    }

    /* ... */

    /* Restore privileges if needed */
    if (geteuid() != privid) {
        if (seteuid(privid) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
```

```
    /* Permanently drop privileges */
    if (setuid(getuid()) != 0) {
        /* Handle error */
        fatal_error();
    }

    if (setuid(0) != -1) {
        /* Privileges can be restored, handle error */
        fatal_error();
    }

    /* Code intended to run with lower privileges; */
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `MISSING_PRIVILEGE_DROP_CHECK`
Impact: High
CWE ID: 250, 273

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2016b

Returned value of a sensitive function not checked

Sensitive functions called without checking for unexpected return values and errors

Description

This defect occurs when you call sensitive standard functions, but you:

- Ignore the return value.
- Use an output or a return value without testing the validity of the return value.

For this defect, two type of functions are considered: ***sensitive*** and ***critical sensitive***.

A ***sensitive*** function is a standard function that can encounter:

- Exhausted system resources (for example, when allocating resources)
- Changed privileges or permissions
- Tainted sources when reading, writing, or converting data from external sources
- Unsupported features despite an existing API

A ***critical sensitive*** function is a sensitive function that performs one of these critical or vulnerable tasks:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

Risk

If you do not check the return value of functions that perform sensitive or critical sensitive tasks, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

Fix

Before continuing with the program, test the return value of *critical sensitive* functions.

For *sensitive functions*, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

Examples

Sensitive Function Return Ignored

```
#include <pthread.h>
```

```
void initialize() {
    pthread_attr_t attr;

    pthread_attr_init(&attr);
}
```

This example shows a call to the sensitive function `pthread_attr_init`. The return value of `pthread_attr_init` is ignored, causing a defect.

Correction — Cast Function to `(void)`

One possible correction is to cast the function to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of the sensitive function.

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    (void)pthread_attr_init(&attr);
}
```

Correction — Test Return Value

One possible correction is to test the return value of `pthread_attr_init` to check for errors.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

void initialize() {
    pthread_attr_t attr;
    int result;

    result = pthread_attr_init(&attr);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

Critical Function Return Ignored

```
#include <pthread.h>
extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0));
    pthread_join(thread_id,  &res);
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a

critical function (not just a sensitive function), Polyspace does not ignore this *Return value of a sensitive function not checked* defect. The other critical function, `pthread_join`, returns value that is ignored implicitly. `pthread_join` uses the return value of `pthread_create`, which was not checked.

Correction — Test the Return Value of Critical Functions

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `RETURN_NOT_CHECKED`
Impact: High
CWE ID: 252, 253, 690, 754

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2016b

Sensitive data printed out

Function prints sensitive data

Description

This defect occurs when print functions such as `stdout` or `stderr` print sensitive information.

The checker considers the following as sensitive information:

- Return values of password manipulation functions such as `getpw`, `getpwnam` or `getpwuid`.
- Input values of functions such as the Windows-specific function `LogonUser`.

Risk

Printing sensitive information, such as passwords or user information, allows an attacker additional access to the information.

Fix

One fix for this defect is to not print out sensitive information.

If you are saving your logfile to an external file, set the file permissions so that attackers cannot access the logfile information.

Examples

Printing Passwords

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

extern void verify_null(const char* buf);
void bug_sensitivedataprint(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    puts("Name\n");
    puts(pwd.pw_name);
    puts("PassWord\n");
    puts(pwd.pw_passwd);
    memset(buf, 0, sizeof(buf));
    verify_null(buf);
}
```

In this example, Bug Finder flags `puts` for printing out the password `pwd.pw_passwd`.

Correction — Obfuscate the Password

One possible correction is to obfuscate the password information so that the information is not visible.

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

extern void verify_null(const char* buf);

void sensitivedataprint(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    puts("Name\n");
    puts(pwd.pw_name);
    puts("PassWord\n");
    puts("XXXXXXXX\n");
    memset(buf, 0, sizeof(buf));
    verify_null(buf);
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `SENSITIVE_DATA_PRINT`
Impact: Medium
CWE ID: 532, 534, 535

See Also

`Sensitive heap memory not cleared before release` | `Uncleared sensitive data in stack` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Sensitive heap memory not cleared before release

Sensitive data not cleared or released by memory routine

Description

This defect occurs when dynamically allocated memory contains sensitive data and you do not clear the data before you free the memory.

Risk

If the memory zone is reallocated, an attacker can still inspect the sensitive data in the old memory zone.

Fix

Before calling `free`, clear out the sensitive data using `memset` or `SecureZeroMemory`.

Examples

Sensitive Buffer Freed, Not Cleared

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>

void sensitiveheapnotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char* buf = (char*) malloc(1024);
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    free(buf);
}
```

In this example, the function uses a buffer of passwords and frees the memory before the end of the function. However, the data in the memory is not cleared by using the `free` command.

Correction — Nullify Data

One possible correction is to write over the data to clear out the sensitive information. This example uses `memset` to write over the data with zeros.

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>
#include <assert.h>

#define isNull(arr) for(int i=0;i<(sizeof(arr)/sizeof(arr[0]));i++) assert(arr[i]==0)

void sensitiveheapnotcleared(const char * my_user) {
```

```
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char* buf = (char*) malloc(1024);

    if (buf) {
        getpwnam_r(my_user, &pwd, buf, bufsize, &result);
        memset(buf, 0, (size_t)1024);
        isNull(buf);
        free(buf);
    }
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: SENSITIVE_HEAP_NOT_CLEARED
Impact: Medium
CWE ID: 244, 312, 316

See Also

`Uncleared sensitive data in stack` | `Sensitive data printed out` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Umask used with chmod-style arguments

Argument to `umask` allows external user too much control

Description

This defect occurs when `umask` commands have arguments specified in the style of arguments to `chmod`.

For new files, the `umask` value specifies which permissions *not* to set, in other words, which permissions to remove. The umask argument is bitwise-negated and then applied to new file permissions. In contrast, chmod sets the permissions as you specify them.

Risk

If you use `chmod`-style arguments, you specify opposite permissions of what you want. This mistake can give external users unintended read/write access to new files and folders.

Fix

Set the umask so that the user (`u`) has fewer permissions turned off than the group (`g`). Set umask so that the group has fewer permissions turned off than other users (`o`), or `u <= g <= o`.

You can see the umask value by calling,

```
umask
```

or the symbolic value by calling,

```
umask -S
```

Examples

Setting the Default Mask

```
#include <stdio.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>

typedef mode_t (*umask_func)(mode_t);

const mode_t default_mode = (
    S_IRUSR    /* 00400 */
    | S_IWUSR  /* 00200 */
    | S_IRGRP  /* 00040 */
    | S_IWGRP  /* 00020 */
    | S_IROTH  /* 00004 */
    | S_IWOTH  /* 00002 */
    );         /* 00666 (i.e. -rw-rw-rw-) */

static void my_umask(mode_t mode)
{
    umask(mode);
```

```
}

int umask_use(mode_t m)
{
    my_umask(default_mode);
    return 0;
}
```

This example uses a function called `my_umask` to set the default mask mode. However, the `default_mode` variable gives the permissions 666 or `-rw-rw-rw-`. `umask` negates this value. However, this negation means the default mask mode turns off read/write permissions for the user, group users, and other outside users.

Correction — Negate Preferred Permissions

One possible correction is to negate the `default_mode` argument to `my_umask`. This correction nullifies the negation `umask` for new files.

```
#include <stdio.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>

typedef mode_t (*umask_func)(mode_t);

const mode_t default_mode = (
    S_IRUSR    /* 00400 */
    | S_IWUSR  /* 00200 */
    | S_IRGRP  /* 00040 */
    | S_IWGRP  /* 00020 */
    | S_IROTH  /* 00004 */
    | S_IWOTH  /* 00002 */
    );         /* 00666 (i.e. -rw-rw-rw-) */

static void my_umask(mode_t mode)
{
    umask(mode);
}

int umask_use(mode_t m)
{
    my_umask(~default_mode);
    return 0;
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `BAD_UMASK`
Impact: Low
CWE ID: 560, 922

See Also

`Vulnerable permission assignments` | `Find defects (-checkers)`

Topics
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

External Websites
`umask` — Linux Manual Page

Introduced in R2015b

Uncleared sensitive data in stack

Variable in stack is not cleared and contains sensitive data

Description

This defect occurs when statically allocated memory contains sensitive data and you do not clear the data before exiting a function or program.

Risk

Leaving sensitive information in your stack, such as passwords or user information, allows an attacker additional access to the information after your program has ended.

Fix

Before exiting a function or program, clear out the memory zones that contain sensitive data by using `memset` or `SecureZeroMemory`.

Examples

Static Buffer of Password Information

```
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>

void bug_sensitivestacknotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
}
```

In this example, a static buffer is filled with password information. The program frees the stack memory at the end of the program. However, the data is still accessible from the memory.

Correction — Clear Memory

One possible correction is to write over the memory before exiting the function. This example uses `memset` to clear the data from the buffer memory.

```
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>
#include <assert.h>

#define isNull(arr) for(int i=0;i<(sizeof(arr)/sizeof(arr[0]));i++) assert(arr[i]==0)

void corrected_sensitivestacknotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
```

```
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    memset(buf, 0, (size_t)1024);
    isNull(buf);
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `SENSITIVE_STACK_NOT_CLEARED`
Impact: Medium
CWE ID: 226, 312, 316

See Also

`Sensitive heap memory not cleared before release` | `Sensitive data printed out` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Unsafe call to a system function

Unsanitized command argument has exploitable vulnerabilities

Description

This defect occurs when you use a function that invokes an implementation-defined command processor. These functions include:

- The C standard `system()` function.
- The POSIX `popen()` function.
- The Windows `_popen()` and `_wpopen()` functions.

Risk

If the argument of a function that invokes a command processor is not sanitized, it can cause exploitable vulnerabilities. An attacker can execute arbitrary commands or read and modify data anywhere on the system.

Fix

Do not use a `system`-family function to invoke a command processor. Instead, use safer functions such as POSIX `execve()` and WinAPI `CreateProcess()`.

Examples

`system()` Called

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};

void func(char *arg)
{
    char buf[SIZE512];
    int retval=sprintf(buf, "/usr/bin/any_cmd %s", arg);

    if (retval<=0 || retval>SIZE512){
        /* Handle error */
        abort();
    }
    /* Use of system() to pass any_cmd with
    unsanitized argument to command processor */

    if (system(buf) == -1) {
    /* Handle error */
  }
}
```

In this example, `system()` passes its argument to the host environment for the command processor to execute. This code is vulnerable to an attack by command-injection.

Correction — Sanitize Argument and Use execve()

In the following code, the argument of `any_cmd` is sanitized, and then passed to `execve()` for execution. `exec`-family functions are not vulnerable to command-injection attacks.

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};


void func(char *arg)
{
  char *const args[SIZE3] = {"any_cmd", arg, NULL};
  char  *const env[] = {NULL};

  /* Sanitize argument */

  /* Use execve() to execute any_cmd. */

  if (execve("/usr/bin/time", args, env) == -1) {
    /* Handle error */
  }
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `UNSAFE_SYSTEM_CALL`
Impact: High
CWE ID: 78, 88

See Also

`Command executed from externally controlled path` | `Execution of externally controlled command` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017b

Unsafe standard encryption function

Function is not reentrant or uses a risky encryption algorithm

Description

This defect occurs when a standard encryption function uses a broken or weak cryptographic algorithm. For example, `crypt` is not reentrant and is based on the risky Data Encryption Standard (DES).

Risk

The use of a broken, weak, or nonstandard algorithm can expose sensitive information to an attacker. A determined hacker can access the protected data using various techniques.

If the weak function is nonreentrant, when you use the function in concurrent programs, there is an additional race condition risk.

Fix

Avoid functions that use these encryption algorithms. Instead, use a reentrant function that uses a stronger encryption algorithm.

Note Some implementations of `crypt` support additional, possibly more secure, encryption algorithms.

Examples

Decrypting Password Using `crypt`

```
#define _GNU_SOURCE
#include <pwd.h>
#include <string.h>
#include <crypt.h>

volatile int rd = 1;

const char *salt = NULL;
struct crypt_data input, output;

int verif_pwd(const char *pwd, const char *cipher_pwd, int safe)
{
    int r = 0;
    char *decrypted_pwd = NULL;

    switch(safe)
    {
      case 1:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      case 2:
```

```
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      default:
        decrypted_pwd = crypt(pwd, cipher_pwd);
        break;
    }

    r = (strcmp(cipher_pwd, decrypted_pwd) == 0);

    return r;
}
```

In this example, `crypt_r` and `crypt` decrypt a password. However, `crypt` is nonreentrant and uses the unsafe Data Encryption Standard algorithm.

Correction — Use `crypt_r`

One possible correction is to replace `crypt` with `crypt_r`.

```
#define _GNU_SOURCE
#include <pwd.h>
#include <string.h>
#include <crypt.h>

volatile int rd = 1;

const char *salt = NULL;
struct crypt_data input, output;

int verif_pwd(const char *pwd, const char *cipher_pwd, int safe)
{
    int r = 0;
    char *decrypted_pwd = NULL;

    switch(safe)
    {
      case 1:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      case 2:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      default:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;
    }

    r = (strcmp(cipher_pwd, decrypted_pwd) == 0);

    return r;
}
```

Result Information

Group: Security

Language: C | C++
Default: Off
Command-Line Syntax: `UNSAFE_STD_CRYPT`
Impact: Medium
CWE ID: 327, 522, 663

See Also

`Deterministic random output from constant seed` | `Predictable random output from predictable seed` | `Vulnerable pseudo-random number generator` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Unsafe standard function

Function unsafe for security-related purposes

Description

This defect occurs when you use standard functions that are unsafe and must not be used for security-related programming. Functions can be unsafe for many reasons. Some functions are unsafe because they are nonreentrant. Other functions change behavior depending on the target or platform, making some implementations unsafe.

Risk

Some unsafe functions are not reentrant, meaning that the contents of the function are not locked during a call. So, an attacker can change the values midstream.

`getlogin` specifically can be unsafe depending on the implementation. Some implementations of `getlogin` return only the first eight characters of a log-in name. An attacker can use a different login with the same first eight characters to gain entry and manipulate the program.

Fix

Avoid unsafe functions for security-related purposes. If you cannot avoid unsafe functions, use a safer version of the function instead. For `getlogin`, use `getlogin_r`.

Examples

Using `getlogin`

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <string.h>
#include <stdlib.h>


volatile int rd = 1;

int login_name_check(char *user)
{
    int r = -2;
    char *name = getlogin();
    if (name != NULL)
    {
        if (strcmp(name, user) == 0)
        {
            r = 0;
        }
        else
            r = -1;
    }
```

```
    return r;
}
```

This example uses `getlogin` to compare the user name of the current user to the given user name . However, `getlogin` can return something other than the current user name because a parallel process can change the string.

Correction — Use `getlogin_r`

One possible correction is to use `getlogin_r` instead of `getlogin`. `getlogin_r` is reentrant, so you can trust the result.

```
#define _POSIX_C_SOURCE 199506L // use of getlogin_r
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <string.h>
#include <stdlib.h>


volatile int rd = 1;

enum {  NAME_MAX_SIZE=64  };

int login_name_check(char *user)
{
    int r;
    char name[NAME_MAX_SIZE];

    if (getlogin_r(name, sizeof(name)) == 0)
    {
        if ((strlen(user) < sizeof(name)) &&
                    (strncmp(name, user, strlen(user)) == 0))
        {
            r = 0;
        }
        else
            r = -1;
    }
    else
        r = -2;
    return r;
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `UNSAFE_STD_FUNC`
Impact: Medium
CWE ID: 558, 663

See Also

`Use of obsolete standard function` | `Use of dangerous standard function` | `Invalid use of standard library string routine` | `Find defects (-checkers)`

Topics
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Use of dangerous standard function

Dangerous functions cause possible buffer overflow in destination buffer

Description

This defect occurs when your code uses functions that are inherently dangerous or potentially dangerous given certain circumstances. The following table lists possibly dangerous functions, the risks of using each function, and what function to use instead.

Dangerous Function	Risk Level	Safer Function
`gets`	Inherently dangerous — You cannot control the length of input from the console.	`fgets`
`cin`	Inherently dangerous — You cannot control the length of input from the console.	Avoid or prefaces calls to `cin` with `cin.width`.
`strcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`strncpy`
`stpcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`stpncpy`
`lstrcpy` or `StrCpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy`
`strcat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`strncat`, `strlcat`, or `strcat_s`
`lstrcat` or `StrCat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`StringCbCat`, `StringCchCat`, `strncay`, `strcat_s`, or `strlcat`
`wcpcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`wcpncpy`
`wcscat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`wcsncat`, `wcslcat`, or `wcncat_s`
`wcscpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`wcsncpy`
`sprintf`	Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur.	`snprintf`

Dangerous Function	Risk Level	Safer Function
`vsprintf`	Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur.	`vsnprintf`

Risk

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Using `sprintf`

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (sprintf(dst, "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

Correction — Use `snprintf` with Buffer Size

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
```

```
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: DANGEROUS_STD_FUNC
Impact: Low
CWE ID: 242, 676

See Also

`Use of obsolete standard function` | `Unsafe standard function` | `Invalid use of standard library string routine` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Use of non-secure temporary file

Temporary generated file name not secure

Description

This defect occurs when you use temporary file routines that are not secure.

Risk

If an attacker guesses the file name generated by a standard temporary file routine, the attacker can:

- Cause a race condition when you generate the file name.
- Precreate a file of the same name, filled with malicious content. If your program reads the file, the attacker's file can inject the malicious code.
- Create a symbolic link to a file storing sensitive data. When your program writes to the temporary file, the sensitive data is deleted.

Fix

To create temporary files, use a more secure standard temporary file routine, such as `mkstemp` from POSIX.1-2001.

Also, when creating temporary files with routines that allow flags, such as `mkostemp`, use the exclusion flag `O_EXCL` to avoid race conditions.

Examples

Temp File Created With `tempnam`

```
#define _BSD_SOURCE
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int test_temp()
{
    char tpl[] = "abcXXXXXX";
    char suff_tpl[] = "abcXXXXXXsuff";
    char *filename = NULL;
    int fd;

    filename = tempnam("/var/tmp", "foo_");

    if (filename != NULL)
    {
        printf("generated tmp name (%s) in (%s:%s:%s)\n",
```

```
               filename, getenv("TMPDIR") ? getenv("TMPDIR") : "$TMPDIR",
               "/var/tmp", P_tmpdir);

        fd = open(filename, O_CREAT, S_IRWXU|S_IRUSR);
        if (fd != -1)
        {
            close(fd);
            unlink(filename);
            return 1;
        }
    }
    return 0;
}
```

In this example, Bug Finder flags `open` because it tries to use an unsecure temporary file. The file is opened without exclusive privileges. An attacker can access the file causing various risks on page 3-392.

Correction — Add `O_EXCL` Flag

One possible correction is to add the `O_EXCL` flag when you open the temporary file.

```
#define _BSD_SOURCE
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int test_temp()
{
    char tpl[] = "abcXXXXXX";
    char suff_tpl[] = "abcXXXXXXsuff";
    char *filename = NULL;
    int fd;

    filename = tempnam("/var/tmp", "foo_");

    if (filename != NULL)
    {
        printf("generated tmp name (%s) in (%s:%s:%s)\n",
               filename, getenv("TMPDIR") ? getenv("TMPDIR") : "$TMPDIR",
               "/var/tmp", P_tmpdir);

        fd = open(filename, O_CREAT|O_EXCL, S_IRWXU|S_IRUSR);
        if (fd != -1)
        {
            close(fd);
            unlink(filename);
            return 1;
        }
    }
    return 0;
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `NON_SECURE_TEMP_FILE`
Impact: High
CWE ID: 377, 922

See Also

`Data race` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Use of obsolete standard function

Obsolete routines can cause security vulnerabilities and portability issues

Description

This defect occurs when you use standard function routines that are considered legacy, removed, deprecated, or obsolete by C/C++ coding standards.

Obsolete Function	Standards	Risk	Replacement Function
`asctime`	Deprecated in POSIX.1-2008	Not thread-safe.	`strftime` or `asctime_s`
`asctime_r`	Deprecated in POSIX.1-2008	Implementation based on unsafe function `sprintf`.	`strftime` or `asctime_s`
`bcmp`	Deprecated in 4.3BSD Marked as legacy in POSIX.1-2001.	Returns from function after finding the first differing byte, making it vulnerable to timing attacks.	`memcmp`
`bcopy`	Deprecated in 4.3BSD Marked as legacy in POSIX.1-2001.	Returns from function after finding the first differing byte, making it vulnerable to timing attacks.	`memcpy` or `memmove`
`brk` and `sbrk`	Marked as legacy in SUSv2 and POSIX.1-2001.		`malloc`
`bsd_signal`	Removed in POSIX.1-2008		`sigaction`
`bzero`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`memset`
`ctime`	Deprecated in POSIX.1-2008	Not thread-safe.	`strftime` or `asctime_s`
`ctime_r`	Deprecated in POSIX.1-2008	Implementation based on unsafe function `sprintf`.	`strftime` or `asctime_s`
`cuserid`	Removed in POSIX.1-2001.	Not reentrant. Precise functionality not standardized causing portability issues.	`getpwuid`
`ecvt` and `fcvt`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008	Not reentrant	`snprintf`
`ecvt_r` and `fcvt_r`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008		`snprintf`
`ftime`	Removed in POSIX.1-2008		`time`, `gettimeofday`, `clock_gettime`

Obsolete Function	Standards	Risk	Replacement Function
`gamma`, `gammaf`, `gammal`	Function not specified in any standard because of historical variations	Portability issues.	`tgamma`, `lgamma`
`gcvt`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`snprintf`
`getcontext`	Removed in POSIX.1-2008.	Portability issues.	Use POSIX thread instead.
`getdtablesize`	BSD API function not included in POSIX.1-2001	Portability issues.	`sysconf( _SC_OPEN _MAX )`
`gethostbyaddr`	Removed in POSIX.1-2008	Not reentrant	`getaddrinfo`
`gethostbyname`	Removed in POSIX.1-2008	Not reentrant	`getnameinfo`
`getpagesize`	BSD API function not included in POSIX.1-2001	Portability issues.	`sysconf( _SC_PAGE SIZE )`
`getpass`	Removed in POSIX.1-2001.	Not reentrant.	`getpwuid`
`getw`	Not present in POSIX.1-2001.		`fread`
`getwd`	Marked legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`getcwd`
`index`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`strchr`
`makecontext`	Removed in POSIX.1-2008.	Portability issues.	Use POSIX thread instead.
`memalign`	Appears in SunOS 4.1.3. Not in 4.4 BSD or POSIX.1-2001		`posix_memalign`
`mktemp`	Removed in POSIX.1-2008.	Generated names are predictable and can cause a race condition.	`mkstemp` removes race risk
`pthread_attr_ getstackaddr` and `pthread_attr_ setstackaddr`		Ambiguities in the specification of the `stackaddr` attribute cause portability issues	`pthread_attr_ getstack` and `pthread_attr_ setstack`
`putw`	Not present in POSIX.1-2001.	Portability issues.	`fwrite`
`qecvt` and `qfcvt`	Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008		`snprintf`
`qecvt_r` and `qfcvt_r`	Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008		`snprintf`
`rand_r`	Marked as obsolete in POSIX.1-2008		
`re_comp`	BSD API function	Portability issues	`regcomp`
`re_exes`	BSD API function	Portability issues	`regexec`
`rindex`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`strrchr`

Obsolete Function	Standards	Risk	Replacement Function
`scalb`	Removed in POSIX.1-2008		`scalbln`, `scalblnf`, or `scalblnl`
`sigblock`	4.3BSD signal API whose origin is unclear		`sigprocmask`
`sigmask`	4.3BSD signal API whose origin is unclear		`sigprocmask`
`sigsetmask`	4.3BSD signal API whose origin is unclear		`sigprocmask`
`sigstack`	Interface is obsolete and not implemented on most platforms.	Portability issues.	`sigaltstack`
`sigvec`	4.3BSD signal API whose origin is unclear		`sigaction`
`swapcontext`	Removed in POSIX.1-2008	Portability issues.	Use POSIX threads.
`tmpnam` and `tmpnam_r`	Marked as obsolete in POSIX.1-2008.	This function generates a different string each time it is called, up to TMP_MAX times. If it is called more than TMP_MAX times, the behavior is implementation-defined.	`mkstemp`, `tmpfile`
`ttyslot`	Removed in POSIX.1-2001.		
`ualarm`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.	Errors are under-specified	`setitimer` or POSIX `timer_create`
`usleep`	Removed in POSIX.1-2008.		`nanosleep`
`utime`	SVr4, POSIX.1-2001. POSIX.1-2008 marks as obsolete.		
`valloc`	Marked as obsolete in 4.3BSD. Marked as legacy in SUSv2. Removed from POSIX.1-2001		`posix_memalign`
`vfork`	Removed from POSIX.1-2008	Under-specified in previous standards.	`fork`
`wcswcs`	This function was not included in the final ISO/IEC 9899:1990/ Amendment 1:1995 (E).		`wcsstr`
`WinExec`	WinAPI provides this function only for 16-bit Windows compatibility.		`CreateProcess`
`LoadModule`	WinAPI provides this function only for 16-bit Windows compatibility.		`CreateProcess`

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Printing Out Time

```
#include <stdio.h>
#include <time.h>

void timecheck_bad(int argc, char *argv[])
{
    time_t ticks;

    ticks = time(NULL);
    printf("%.24s\r\n", ctime(&ticks));
}
```

In this example, the function `ctime` formats the current time and prints it out. However, `ctime` was removed after C99 because it does not work on multithreaded programs.

Correction — Different Time Function

One possible correction is to use `strftime` instead because this function uses a set buffer size.

```
#include <stdio.h>
#include <string.h>
#include <time.h>

void timecheck_good(int argc, char *argv[])
{
    char outBuff[1025];
    time_t ticks;
    struct tm * timeinfo;

    memset(outBuff, 0, sizeof(outBuff));

    ticks = time(NULL);
    timeinfo = localtime(&ticks);
    strftime(outBuff,sizeof(outBuff),"%I:%M%p.",timeinfo);
    fprintf(stdout, outBuff);
}
```

Result Information

Group: Security
Language: C | C++

Default: Off
Command-Line Syntax: `OBSOLETE_STD_FUNC`
Impact: Low
CWE ID: 474, 477
Tags: #deprecatedFunctions

See Also

`Use of dangerous standard function` | `Unsafe standard function` | `Invalid use of standard library string routine` | `Find defects (-checkers)`

Topics
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Vulnerable path manipulation

Path argument with `/../`, `/abs/path/`, or other unsecure elements

Description

This defect occurs when you create a relative or absolute path from a tainted source and you then use the path to open/create files.

Risk

Relative path elements, such as `".."` can resolve to locations outside the intended folder. Absolute path elements, such as `"/abs/path"` can also resolve to locations outside the intended folder.

An attacker can use these types of path traversal elements to traverse to the rest of the file system and access other files or folders.

Fix

Avoid vulnerable path traversal elements such as `/../` and `/abs/path/`. Use fixed file names and locations wherever possible.

Examples

Relative Path Traversal

```
# include <stdio.h>
# include <string.h>
# include <wchar.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <unistd.h>
# include <stdlib.h>
# define BASEPATH "/tmp/"
# define FILENAME_MAX 512

static void Relative_Path_Traversal(void)
{
    char * data;
    char data_buf[FILENAME_MAX] = BASEPATH;
    char sub_buf[FILENAME_MAX];

    if (fgets(sub_buf, FILENAME_MAX, stdin) == NULL) exit (1);
    data = data_buf;
    strcat(data, sub_buf);

    FILE *file = NULL;
    file = fopen(data, "wb+");
    if (file != NULL) fclose(file);
}

int path_call(void){
```

```
    Relative_Path_Traversal();
}
```

This example opens a file from `"/tmp/"`, but uses a relative path to the file. An external user can manipulate this relative path when `fopen` opens the file.

Correction — Use Fixed File Name

One possible correction is to use a fixed file name instead of a relative path. This example uses `file.txt`.

```
# include <stdio.h>
# include <string.h>
# include <wchar.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <unistd.h>
# include <stdlib.h>
# define BASEPATH "/tmp/"
# define FILENAME_MAX 512

static void Relative_Path_Traversal(void)
{
    char * data;
    char data_buf[FILENAME_MAX] = BASEPATH;
    data = data_buf;

    /* FIX: Use a fixed file name */
    strcat(data, "file.txt");
    FILE *file = NULL;
    file = fopen(data, "wb+");
    if (file != NULL) fclose(file);
}

int path_call(void){
    Relative_Path_Traversal();
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `PATH_TRAVERSAL`
Impact: Low
CWE ID: 22, 23, 36

See Also

`Use of path manipulation function without maximum sized buffer checking` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Vulnerable permission assignments

Argument gives read/write/search permissions to external users

Description

This defect occurs when functions that can change resource permissions, such as `chmod`, `umask`, `creat`, or `open`, specify permissions that allow unintended actors to modify or read the resource.

Risk

If you give outside users or outside groups a wider range or permissions than required, you potentially expose your sensitive information and your modifications. This defect is especially dangerous for permissions related to:

- Program configurations
- Program executions
- Sensitive user data

Fix

Set your permissions so that the user (`u`) has more permissions than the group (`g`), and so the group has more permissions than other users (`o`), or `u >= g >= o`.

Examples

Create File with Other Permissions

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void bug_dangerouspermissions(const char * log_path) {
    mode_t mode = S_IROTH | S_IXOTH | S_IWOTH;
    int fd = creat(log_path, mode);

    if (fd) {
        write(fd, "Hello\n", 6);
    }
    close(fd);
    unlink(log_path);
}
```

In this example, the `log_path` file is created with more rights for the other outside users, than the current user. The permissions are `---------rwx`.

Correction — Modify User Permissions

One possible correction is to modify the user permissions for the file. In this correction, the user has read/write/execute permissions, but other users do not.

```
#include <unistd.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>

void corrected_dangerouspermissions(const char * log_path) {
    mode_t mode = S_IRUSR | S_IXUSR | S_IWUSR;
    int fd = creat(log_path, mode);

    if (fd) {
        write(fd, "Hello\n", 6);
    }
    close(fd);
    unlink(log_path);
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: DANGEROUS_PERMISSIONS
Impact: Medium
CWE ID: 732, 922

See Also

`Umask used with chmod-style arguments` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Vulnerable pseudo-random number generator

Using a cryptographically weak pseudo-random number generator

Description

This defect occurs when you use cryptographically weak pseudo-random number generator (PRNG) routines.

The list of cryptographically weak routines flagged by this checker include:

- `rand`, `random`
- `drand48`, `lrand48`, `mrand48`, `erand48`, `nrand48`, `jrand48`, and their `_r` equivalents such as `drand48_r`
- `RAND_pseudo_bytes`

Risk

These cryptographically weak routines are predictable and must not be used for security purposes. When a predictable random value controls the execution flow, your program is vulnerable to malicious attacks.

Fix

Use more cryptographically sound random number generators, such as `CryptGenRandom` (Windows), `OpenSSL/RAND_bytes`(Linux/UNIX).

Examples

Random Loop Numbers

```
#include <stdio.h>
#include <stdlib.h>

volatile int rd = 1;
int main(int argc, char *argv[])
{
    int j, r, nloops;
    struct random_data buf;
    int i = 0;

    nloops = rand();

    for (j = 0; j < nloops; j++) {
        if (random_r(&buf, &i))
            exit(1);
        printf("random_r: %ld\n", (long)i);
    }
    return 0;
}
```

This example uses `rand` and `random_r` to generate random numbers. If you use these functions for security purposes, these PRNGs can be the source of malicious attacks.

Correction — Use Stronger PRNG

One possible correction is to replace the vulnerable PRNG with a stronger random number generator.

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/rand.h>

volatile int rd = 1;
int main(int argc, char* argv[])
{
    int j, r, nloops;
    unsigned char buf;
    unsigned int seed;
    int i = 0;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <seed> <nloops>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    seed = atoi(argv[1]);
    nloops = atoi(argv[2]);

    for (j = 0; j < nloops; j++) {
        if (RAND_bytes(&buf, i) != 1)
            exit(1);
        printf("RAND_bytes: %u\n", (unsigned)buf);
    }
    return 0;
}
```

Result Information

Group: Security
Language: C | C++
Default: Off
Command-Line Syntax: `VULNERABLE_PRNG`
Impact: Medium
CWE ID: 330, 338

See Also

`Deterministic random output from constant seed` | `Predictable random output from predictable seed` | `Unsafe standard encryption function` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Cryptography Defects

Constant block cipher initialization vector

Initialization vector is constant instead of randomized

Description

This defect occurs when you use a constant for the initialization vector (IV) during encryption.

Risk

Using a constant IV is equivalent to not using an IV. Your encrypted data is vulnerable to dictionary attacks.

Block ciphers break your data into blocks of fixed size. Block cipher modes such as CBC (Cipher Block Chaining) protect against dictionary attacks by XOR-ing each block with the encrypted output from the previous block. To protect the first block, these modes use a random initialization vector (IV). If you use a constant IV to encrypt multiple data streams that have a common beginning, your data becomes vulnerable to dictionary attacks.

Fix

Produce a random IV by using a strong random number generator.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

Examples

Constants Used for Initialization Vector

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

/* Using the cryptographic routines */

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16] = {'1', '2', '3', '4','5','6','b','8','9',
                                '1','2','3','4','5','6','7'};
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

In this example, the initialization vector `iv` has constants only. The constant initialization vector makes your cipher vulnerable to dictionary attacks.

Correction — Use Random Initialization Vector

One possible correction is to use a strong random number generator to produce the initialization vector. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

/* Using the cryptographic routines */

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16];
    RAND_bytes(iv, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_CIPHER_CONSTANT_IV`
Impact: Medium
CWE ID: 310, 326, 329

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Constant cipher key

Encryption or decryption key is constant instead of randomized

Description

This defect occurs when you use a constant for the encryption or decryption key.

Risk

If you use a constant for the encryption or decryption key, an attacker can retrieve your key easily.

You use a key to encrypt and later decrypt your data. If a key is easily retrieved, data encrypted using that key is not secure.

Fix

Produce a random key by using a strong random number generator.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

Examples

Constants Used for Key

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16] = {'1', '2', '3', '4','5','6','b','8','9',
                                 '1','2','3','4','5','6','7'};
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

In this example, the cipher key, `key`, has constants only. An attacker can easily retrieve a constant key.

Correction — Use Random Key

Use a strong random number generator to produce the cipher key. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
```

```
    unsigned char key[SIZE16];
    RAND_bytes(key, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_CIPHER_CONSTANT_KEY`
Impact: Medium
CWE ID: 310, 320, 321, 326, 522

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Context initialized incorrectly for cryptographic operation

Context used for public key cryptography operation is initialized for a different operation

Description

This defect occurs when you initialize an EVP_PKEY_CTX object for a specific public key cryptography operation but use the object for a different operation.

For instance, you initialize the context for encryption.

```
ret = EVP_PKEY_encrypt_init(ctx);
```

However, you use the context for decryption without reinitializing the context.

```
ret = EVP_PKEY_decrypt(ctx, out, &out_len, in, in_len);
```

The checker detects if the context object used in these functions has been initialized by using the corresponding initialization functions: EVP_PKEY_paramgen, EVP_PKEY_keygen, EVP_PKEY_encrypt, EVP_PKEY_verify, EVP_PKEY_verify_recover,EVP_PKEY_decrypt, EVP_PKEY_sign, EVP_PKEY_derive,and EVP_PKEY_derive_set_peer.

Risk

Mixing up different operations on the same context can lead to obscure code. It is difficult to determine at a glance whether the current object is used for encryption, decryption, signature, or another operation. The mixup can also lead to a failure in the operation or unexpected ciphertext.

Fix

After you set up a context for a certain family of operations, use the context for only that family of operations.For instance, use these pairs of functions for initialization and usage of the EVP_PKEY_CTX context object.

- For encryption with EVP_PKEY_encrypt, initialize the context with EVP_PKEY_encrypt_init.
- For signature verification with EVP_PKEY_verify, initialize the context with EVP_PKEY_verify_init.
- For key generation with EVP_PKEY_keygen, initialize the context with EVP_PKEY_keygen_init.

If you want to reuse an existing context object for a different family of operations, reinitialize the context.

Examples

Encryption Using Context Initialized for Decryption

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)
```

```
int ret;
unsigned char *out_buf10;
size_t out_len10;
int func(unsigned char *src, size_t len, EVP_PKEY_CTX *ctx){
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_decrypt_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_encrypt(ctx, out_buf10, &out_len10, src, len);
}
```

In this example, the context is initialized for decryption but used for encryption.

Correction — Use One Family of Operations

One possible correction is to initialize the object for encryption.

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf10;
size_t out_len10;
int func(unsigned char *src, size_t len, EVP_PKEY_CTX *ctx){
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_encrypt_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_encrypt(ctx, out_buf10, &out_len10, src, len);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_PKEY_INCORRECT_INIT`
Impact: Medium
CWE ID: 310, 325, 372, 573, 664

See Also

`Find defects (-checkers)` | `Incorrect key for cryptographic algorithm` | `Missing parameters for key generation` | `Missing data for encryption, decryption or signing operation` | `Missing peer key` | `Missing private key` | `Missing public key` | `Nonsecure parameters for key generation`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Context initialized incorrectly for digest operation

Context used for digest operation is initialized for a different digest operation

Description

This defect occurs when you initialize an `EVP_MD_CTX` context object for a specific digest operation but use the context for a different operation.

For instance, you initialize the context for creating a message digest only.

```
ret = EVP_DigestInit(ctx, EVP_sha256())
```

However, you perform a final step for signing:

```
ret = EVP_SignFinal(&ctx, out, &out_len, pkey);
```

The error is shown only if the final step is not consistent with the initialization of the context. If the intermediate update steps are inconsistent, it does not trigger an error because the intermediate steps do not depend on the nature of the operation. For instance, `EVP_DigestUpdate` works identically to `EVP_SignUpdate`.

Risk

Mixing up different operations on the same context can lead to obscure code. It is difficult to determine at a glance whether the current object is used for message digest creation, signing, or verification. The mixup can also lead to a failure in the operation or unexpected message digest.

Fix

After you set up a context for a certain family of operations, use the context for only that family of operations. For instance, use these pairs of functions for initialization and final steps.

- `EVP_DigestInit` : `EVP_DigestFinal`
- `EVP_DigestInit_ex` : `EVP_DigestFinal_ex`
- `EVP_DigestSignInit` : `EVP_DigestSignFinal`

If you want to reuse an existing context object for a different family of operations, reinitialize the context.

Examples

Inconsistent Initial and Final Digest Operation

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf16;
unsigned int out_len16;
```

```
void func(unsigned char *src, size_t len){
  EVP_MD_CTX* ctx = EVP_MD_CTX_create();

  ret = EVP_SignInit_ex(ctx, EVP_sha256(), NULL);
  if (ret != 1) fatal_error();

  ret = EVP_SignUpdate(ctx, src, len);
  if (ret != 1) fatal_error();

  ret = EVP_DigestSignFinal(ctx, out_buf16, (size_t*) out_len16);

  if (ret != 1) fatal_error();
}
```

In this example, the context object is initialized for signing only with `EVP_SignInit` but the final step attempts to create a signed digest with `EVP_DigestSignFinal`.

Correction — Use One Family of Operations

One possible correction is to use the context object for signing only. Change the final step to `EVP_SignFinal` in keeping with the initialization step.

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf16;
unsigned int out_len16;

void corrected_cryptomdbadfunction(unsigned char *src, size_t len, EVP_PKEY* pkey){
  EVP_MD_CTX* ctx = EVP_MD_CTX_create();

  ret = EVP_SignInit_ex(ctx, EVP_sha256(), NULL);
  if (ret != 1) fatal_error();

  ret = EVP_SignUpdate(ctx, src, len);
  if (ret != 1) fatal_error();

  ret = EVP_SignFinal(ctx, out_buf16, &out_len16, pkey);
  if (ret != 1) fatal_error();
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_MD_BAD_FUNCTION`
Impact: Medium
CWE ID: 310, 353, 354, 372, 573, 664

See Also

`Find defects (-checkers)` | `Nonsecure hash algorithm`

Topics
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Incompatible padding for RSA algorithm operation

Cryptography operation is not supported by the padding type set in context

Description

This defect occurs when you perform an RSA algorithm operation on a context object that is not compatible with the padding previously associated with the object.

For instance, you associate the OAEP padding scheme with a context object but later use the context for signature verification, an operation that the padding scheme does not support.

```
ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_PADDING);
...
ret = EVP_PKEY_verify(ctx, out, out_len, in, in_len);
```

Risk

Padding schemes remove determinism from the RSA algorithm and protect RSA operations from certain kinds of attack.

When you use an incorrect padding scheme, the RSA operation can fail or result in unexpected ciphertext.

Fix

Before performing an RSA operation, associate the context object with a padding scheme that is compatible with the operation.

- Encryption: Use the OAEP padding scheme.

  For instance, use the `EVP_PKEY_CTX_set_rsa_padding` function with the argument `RSA_PKCS1_OAEP_PADDING` or the `RSA_padding_add_PKCS1_OAEP` function.

  ```
  ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_PADDING);
  ```

  You can also use the PKCS#1v1.5 or SSLv23 schemes. Be aware that these schemes are considered insecure.

  You can then use functions such as `EVP_PKEY_encrypt` / `EVP_PKEY_decrypt` or `RSA_public_encrypt` / `RSA_private_decrypt` on the context.
- Signature: Use the RSA-PSS padding scheme.

  For instance, use the `EVP_PKEY_CTX_set_rsa_padding` function with the argument `RSA_PKCS1_PSS_PADDING`.

  ```
  ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_PSS_PADDING);
  ```

  You can also use the ANSI X9.31, PKCS#1v1.5, or SSLv23 schemes. Be aware that these schemes are considered insecure.

  You can then use functions such as the `EVP_PKEY_sign`-`EVP_PKEY_verify` pair or the `RSA_private_encrypt`-`RSA_public_decrypt` pair on the context.

If you perform two kinds of operation with the same context, after the first operation, reset the padding scheme in the context before the second operation.

Examples

OAEP Padding for Signature Operation

```
#include <stddef.h>
#include <openssl/rsa.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;

int func(unsigned char *src, size_t len, RSA* rsa){
  if (rsa == NULL) fatal_error();
  return RSA_private_encrypt(len, src, out_buf, rsa, RSA_PKCS1_OAEP_PADDING);
}
```

In this example, the function `RSA_private_encrypt` performs a signature operation by using the OAEP padding scheme, which supports encryption operations only.

Correction — Use Padding Scheme That Supports Signature

One possible correction is to use the RSA-PSS padding scheme. The corrected example uses the function `RSA_padding_add_PKCS1_PSS` to associate the padding scheme with the context.

```
#include <stddef.h>
#include <openssl/evp.h>
#include <openssl/rsa.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *msg_pad;
unsigned char *out_buf;

int func(unsigned char *src, size_t len, RSA* rsa){
  if (rsa == NULL) fatal_error();

  ret = RSA_padding_add_PKCS1_PSS(rsa, msg_pad, src, EVP_sha256(), -2);
  if (ret <= 0) fatal_error();

  return RSA_private_encrypt(len, msg_pad, out_buf, rsa, RSA_NO_PADDING);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_RSA_BAD_PADDING`
Impact: Medium

CWE ID: 310, 372, 573, 664

See Also

`Find defects (-checkers)|Missing blinding for RSA algorithm|Missing padding for RSA algorithm|Nonsecure RSA public exponent|Weak padding for RSA algorithm`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Inconsistent cipher operations

You perform encryption and decryption steps in succession with the same cipher context without a reinitialization in between

Description

This defect occurs when you perform an encryption and decryption step with the same cipher context. You do not reinitialize the context in between those steps. The checker applies to symmetric encryption only.

For instance, you set up a cipher context for decryption using `EVP_DecryptInit_ex`.

```
EVP_DecryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);
```

However, you use the context for encryption using `EVP_EncryptUpdate`.

```
EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
```

Risk

Mixing up encryption and decryption steps can lead to obscure code. It is difficult to determine at a glance whether the current cipher context is used for encryption or decryption. The mixup can also lead to race conditions, failed encryption, and unexpected ciphertext.

Fix

After you set up a cipher context for a certain family of operations, use the context for only that family of operations.

For instance, if you set up a cipher context for decryption using `EVP_DecryptInit_ex`, use the context afterward for decryption only.

Examples

Encryption Step Following Decryption Step

```
#include <openssl/evp.h>
#include <stdlib.h>

/* Using the cryptographic routines */

unsigned char *out_buf;
int out_len;
unsigned char g_key[16];
unsigned char g_iv[16];
void func(unsigned char* src, int len) {

    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);
```

```
    /* Cipher context set up for decryption*/
    EVP_DecryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, g_key, g_iv);

    /* Update step for encryption */
    EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
}
```

In this example, the cipher context `ctx` is set up for decryption using `EVP_DecryptInit_ex`. However, immediately afterward, the context is used for encryption using `EVP_EncryptUpdate`.

Correction — Change Setup Step

One possible correction is to change the setup step. If you want to use the cipher context for encryption, set it up using `EVP_EncryptInit_ex`.

```
#include <openssl/evp.h>
#include <stdlib.h>

unsigned char *out_buf;
int out_len;
unsigned char g_key[16];
unsigned char g_iv[16];

void func(unsigned char* src, int len) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);

    /* Cipher context set up for encryption*/
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, g_key, g_iv);

    /* Update step for encryption */
    EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_CIPHER_BAD_FUNCTION`
Impact: Medium
CWE ID: 372, 664

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Incorrect key for cryptographic algorithm

Public key cryptography operation is not supported by the algorithm used in context initialization

Description

This defect occurs when you initialize a context object with a key for a specific algorithm but perform an operation that the algorithm does not support.

For instance, you initialize the context with a key for the DSA algorithm.

```
ret = EVP_PKEY_set1_DSA(pkey,dsa);
ctx = EVP_PKEY_CTX_new(pkey, NULL);
```

However, you use the context for encrypting data, an operation that the DSA algorithm does not support.

```
ret = EVP_PKEY_encrypt(ctx,out, &out_len, in, in_len);
```

Risk

If the algorithm does not support your cryptographic operation, you do not see the expected results. For instance, if you use the DSA algorithm for encryption, you might get unexpected ciphertext.

Fix

Use the algorithm that is appropriate for the cryptographic operation that you want to perform:

- Diffie-Hellman (DH): For key derivation.
- Digital Signature Algorithm (DSA): For signature.
- RSA: For encryption and signature.
- Elliptic curve (EC): For key derivation and signature.

Examples

Encryption with DSA Algorithm

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
size_t out_len;

int func(unsigned char *src, size_t len, DSA * dsa){
  EVP_PKEY_CTX *ctx;
  EVP_PKEY *pkey = NULL;

  pkey = EVP_PKEY_new();
  if(pkey == NULL) fatal_error();
```

```
  ret = EVP_PKEY_set1_DSA(pkey,dsa);
  if (ret <= 0) fatal_error();

  ctx = EVP_PKEY_CTX_new(pkey, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_encrypt_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_encrypt(ctx, out_buf, &out_len, src, len);
}
```

In this example, the context object is initialized with a key associated with the DSA algorithm. However, the object is used for encryption, an operation that the DSA algorithm does not support.

Correction — Use RSA Algorithm

One possible correction is to initialize the context object with a key associated with the RSA algorithm.

```
#include <openssl/evp.h>
#include <openssl/rsa.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
size_t out_len;

int func(unsigned char *src, size_t len, RSA * rsa){
  EVP_PKEY_CTX *ctx;
  EVP_PKEY *pkey = NULL;

  pkey = EVP_PKEY_new();
  if(pkey == NULL) fatal_error();

  ret = EVP_PKEY_set1_RSA(pkey,rsa);
  if (ret <= 0) fatal_error();

  ctx = EVP_PKEY_CTX_new(pkey, NULL); /* RSA key is set in the context */
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_encrypt_init(ctx); /* Encryption operation is set in the context */
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_PADDING);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_encrypt(ctx, out_buf, &out_len, src, len);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: CRYPTO_PKEY_INCORRECT_KEY
Impact: Medium
CWE ID: 310, 325, 573, 664

See Also

`Context initialized incorrectly for cryptographic operation` | `Find defects (-checkers)` | `Missing parameters for key generation` | `Missing data for encryption, decryption or signing operation` | `Missing peer key` | `Missing private key` | `Missing public key` | `Nonsecure parameters for key generation`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Missing blinding for RSA algorithm

Context used in decryption or signature verification is not blinded against timing attacks

Description

This defect occurs when you do not enable blinding for an RSA context object before using the object for decryption or signature verification.

For instance, you do not turn on blinding in the context object `rsa` before this decryption step:

```
  ret = RSA_public_decrypt(in_len, in, out, rsa, RSA_PKCS1_PADDING)
```

Risk

Without blinding, the time it takes for the cryptographic operation to be completed has a correlation with the key value. An attacker can gather information about the RSA key by measuring the time for completion. Blinding removes this correlation and protects the decryption or verification operation against timing attacks.

Fix

Before performing RSA decryption or signature verification, enable blinding.

```
ret = RSA_blinding_on(rsa, NULL);
```

Examples

Blinding Disabled Before Decryption

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
int func(unsigned char *src, size_t len, RSA* rsa){
  if (rsa == NULL) fatal_error();

  RSA_blinding_off(rsa);
  return RSA_private_decrypt(len, src, out_buf, rsa, RSA_PKCS1_OAEP_PADDING);
}
```

In this example, blinding is disabled for the context object `rsa`. Decryption with this context object can be vulnerable to timing attacks.

Correction — Enable Blinding Before Decryption

One possible correction is to explicitly enable blinding before decryption. Even if blinding might be enabled previously or by default, explicitly enabling blinding ensures that the security of the current decryption step is not reliant on the caller of `func`.

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
int func(unsigned char *src, size_t len, RSA* rsa){
  if (rsa == NULL) fatal_error();

  ret = RSA_blinding_on(rsa, NULL);
  if (ret <= 0) fatal_error();
  return RSA_private_decrypt(len, src, out_buf, rsa, RSA_PKCS1_OAEP_PADDING);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_RSA_NO_BLINDING`
Impact: Medium
CWE ID: 310, 326, 573

See Also

`Find defects (-checkers)` | `Incompatible padding for RSA algorithm operation` | `Missing padding for RSA algorithm` | `Nonsecure RSA public exponent` | `Weak padding for RSA algorithm`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Missing block cipher initialization vector

Context used for encryption or decryption is associated with NULL initialization vector or not associated with an initialization vector

Description

This defect occurs when you encrypt or decrypt data using a NULL initialization vector (IV).

Note You can initialize your cipher context with a NULL initialization vector (IV). However, if your algorithm requires an IV, before the encryption or decryption step, you must associate the cipher context with a non-NULL IV.

Risk

Many block cipher modes use an initialization vector (IV) to prevent dictionary attacks. If you use a NULL IV, your encrypted data is vulnerable to such attacks.

Block ciphers break your data into blocks of fixed size. Block cipher modes such as CBC (Cipher Block Chaining) protect against dictionary attacks by XOR-ing each block with the encrypted output from the previous block. To protect the first block, these modes use a random initialization vector (IV). If you use a NULL IV, you get the same ciphertext when encrypting the same plaintext. Your data becomes vulnerable to dictionary attacks.

Fix

Before your encryption or decryption steps

```
 ret = EVP_EncryptUpdate(&ctx, out_buf, &out_len, src, len)
```

associate your cipher context `ctx` with a non-NULL initialization vector.

```
ret = EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv)
```

Examples

NULL Initialization Vector Used for Encryption

```
#include <openssl/evp.h>
#include <stdlib.h>
#define fatal_error() abort()

unsigned char *out_buf;
int out_len;

int func(EVP_CIPHER_CTX *ctx, unsigned char *key, unsigned char *src, int len){
    if (key == NULL)
        fatal_error();

    /* Last argument is initialization vector */
```

```
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, NULL);

    /* Update step with NULL initialization vector */
    return EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
}
```

In this example, the initialization vector associated with the cipher context `ctx` is NULL. If you use this context to encrypt your data, your data is vulnerable to dictionary attacks.

Correction — Use Random Initialization Vector

Use a strong random number generator to produce the initialization vector. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define fatal_error() abort()
#define SIZE16 16

unsigned char *out_buf;
int out_len;

int func(EVP_CIPHER_CTX *ctx, unsigned char *key, unsigned char *src, int len){
    if (key == NULL)
        fatal_error();
    unsigned char iv[SIZE16];
    RAND_bytes(iv, 16);

    /* Last argument is initialization vector */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);

    /* Update step with non-NULL initialization vector */
    return EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_CIPHER_NO_IV`
Impact: Medium
CWE ID: 310, 326, 329

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Missing certification authority list

Certificate for authentication cannot be trusted

Description

This defect occurs when you use a context to handle TLS/SSL connections with these functions, but you do not load a certification authority (CA) list into the context.

- `SSL_connect`
- `SSL_accept`
- `SSL_do_handshake`
- `SSL_write`
- `SSL_read`
- `BIO_do_connect`
- `BIO_do_accept`
- `BIO_do_handshake`

A CA is a trusted third party entity that issues digital certificates to other entities. The certificate contains information about its owner. Server or clients use this information to authenticate connections to the certificate owner.

The checker raises a defect if:

- For server authentication, the client has no CA list to determine whether the server certificate is from a trusted source.
- For client authentication, the server has no CA list to determine whether the client certificate is from a trusted source.

Risk

Without a CA list, you cannot determine if the certificate is issued by a trusted CA. The entity that presents the certificate for authentication might not be the entity described in the certificate. Your connection is vulnerable to man-in-the-middle (MITM) attacks.

Fix

Load a certification authority list into the context you create to handle TLS/SSL connections.

Examples

Missing CA List When `SSL_connect` Initiates TLS/SSL Handshake

```
#include <openssl/ssl.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <openssl/err.h>

unsigned char* buf;
```

```
int OpenConnection(char* hostname, int port)
{
    /* Open the connection */
}

SSL_CTX* InitCTX(void)
{
    SSL_CTX* ctx;
    OpenSSL_add_all_algorithms();
    ctx = SSL_CTX_new(TLSv1_2_client_method());
    if (ctx == NULL) {
        /*handle errors */
    }
    return ctx;
}

void func()
{
    SSL_CTX* ctx;
    int server;
    SSL* ssl;
    char buf[1024];
    int bytes;
    char* hostname, *portnum;
    int ret;

    SSL_library_init();
    hostname = "localhost";
    portnum = "4433";

    ctx = InitCTX();
    server = OpenConnection(hostname, atoi(portnum));
    ssl = SSL_new(ctx);
    SSL_set_fd(ssl, server);
    ret = SSL_connect(ssl);
    if (SSL_get_error(ssl, ret) <= 0) {
        char* msg = "Hello???";
        printf("Connected with %s encryption\n", SSL_get_cipher(ssl));
        SSL_write(ssl, msg, strlen(msg));
        bytes = SSL_read(ssl, buf, sizeof(buf));
        buf[bytes] = 0;
        printf("Received: \"%s\"\n", buf);
        SSL_free(ssl);
    } else
        ERR_print_errors_fp(stderr);
    close(server);
    SSL_CTX_free(ctx);
}
```

In this example, a context `ctx` is initialized to handle TLS/SSL connections. When `SSL_connect` initializes the TLS/SSL handshake with the server by using the SSL structure `ssl` created from `ctx`, there is no CA list to check the validity of the server certificate.

Correction — Before Initiating the TLS/SSL Handshake, Load a CA List into the Context

One possible correction is to, before you initialize the SSL structure, specify a list of CA certificates for the context `ctx`, for instance with `SSL_CTX_load_verify_locations`.

```
#include <openssl/ssl.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <openssl/err.h>

unsigned char* buf;

int OpenConnection(char* hostname, int port)
{
    /* Open the connection */
}

SSL_CTX* InitCTX(void)
{
    SSL_CTX* ctx;
    OpenSSL_add_all_algorithms();
    ctx = SSL_CTX_new(TLSv1_2_client_method());
    if (ctx == NULL) {
        /*handle errors */
    }
    return ctx;
}

void LoadCA(SSL_CTX* ctx, char* CertFile, char* CertPath)
{
    if (SSL_CTX_load_verify_locations(ctx, CertFile, CertPath) <= 0) {
        /* handle errors */
    }
}

void func()
{
    SSL_CTX* ctx;
    int server;
    SSL* ssl;
    char buf[1024];
    int bytes;
    char* hostname, *portnum;
    int ret;

    SSL_library_init();
    hostname = "localhost";
    portnum = "4433";

    ctx = InitCTX();
    LoadCA(ctx, "cacert.pem", "ca/");
    server = OpenConnection(hostname, atoi(portnum));
    ssl = SSL_new(ctx);
    SSL_set_fd(ssl, server);
    ret = SSL_connect(ssl);
    if (SSL_get_error(ssl, ret) <= 0) {
        char* msg = "Hello???";
        printf("Connected with %s encryption\n", SSL_get_cipher(ssl));
        SSL_write(ssl, msg, strlen(msg));
        bytes = SSL_read(ssl, buf, sizeof(buf));
        buf[bytes] = 0;
        printf("Received: \"%s\"\n", buf);
```

```
        SSL_free(ssl);
    } else
        ERR_print_errors_fp(stderr);
    close(server);
    SSL_CTX_free(ctx);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_SSL_NO_CA`
Impact: Medium
CWE ID: 310

See Also

`Find defects (-checkers)|Missing X.509 certificate`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019b

Missing cipher algorithm

An encryption or decryption algorithm is not associated with the cipher context

Description

This defect occurs when you do not assign a cipher algorithm when setting up your cipher context.

You can initialize your cipher context without an algorithm. However, before you encrypt or decrypt your data, you must associate the cipher context with a cipher algorithm.

Risk

A missing cipher algorithm can lead to run-time errors or at least, non-secure ciphertext.

Before encryption or decryption, you set up a cipher context that has the information required for encryption: the cipher algorithm and mode, an encryption or decryption key and an initialization vector (for modes that require initialization vectors).

```
ret = EVP_EncryptInit(&ctx, EVP_aes_128_cbc(), key, iv)
```

The function `EVP_aes_128_cbc()` specifies that the Advanced Encryption Standard (AES) algorithm must be used for encryption. The function also specifies a block size of 128 bits and the Cipher Bloch Chaining (CBC) mode.

Instead of specifying the algorithm, you can use NULL in the initialization step. However, before using the cipher context for encryption or decryption, you must perform an additional initialization that associates an algorithm with the context. Otherwise, the update steps for encryption or decryption can lead to run-time errors.

Fix

Before your encryption or decryption steps

```
 ret = EVP_EncryptUpdate(&ctx, out_buf, &out_len, src, len)
```

associate your cipher context `ctx` with an algorithm.

```
ret = EVP_EncryptInit(ctx, EVP_aes_128_cbc(), key, iv)
```

Examples

Algorithm Missing During Context Initialization

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

unsigned char key[SIZE16];
unsigned char iv[SIZE16];
void func(void) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
```

```
    EVP_CIPHER_CTX_init(ctx);
    EVP_EncryptInit_ex(ctx, NULL, NULL, key, iv);
}
```

In this example, an algorithm is not provided when the cipher context `ctx` is initialized.

Before you encrypt or decrypt your data, you have to provide a cipher algorithm. If you perform a second initialization to provide the algorithm, the cipher context is completely re-initialized. Therefore, the current initialization statement using `EVP_EncryptInit_ex` is redundant.

Correction — Provide Algorithm During Initialization

One possible correction is to provide an algorithm when you initialize the cipher context. In the corrected code below, the routine `EVP_aes_128_cbc` invokes the Advanced Encryption Standard (AES) algorithm. The routine also specifies a block size of 128 bits and the Cipher Block Chaining (CBC) mode for encryption.

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

unsigned char key[SIZE16];
unsigned char iv[SIZE16];
void func(unsigned char *src, int len, unsigned char *out_buf, int out_len) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);

    /* Initialization of cipher context */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);

    /* Update steps for encryption */
    EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_CIPHER_NO_ALGORITHM`
Impact: Medium
CWE ID: 310, 573

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Missing cipher data to process

Final encryption or decryption step is performed without previous update steps

Description

This defect occurs when you perform the final step of a block cipher encryption or decryption incorrectly.

For instance, you do one of the following:

- You do not perform update steps for encrypting or decrypting the data before performing a final step.

```
/* Initialization of cipher context */
ret = EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);
...
/* Missing update step */
...
/* Final step */
ret = EVP_EncryptFinal_ex(ctx, out_buf, &out_len);
```

- You perform consecutive final steps without intermediate initialization and update steps.

```
/* Initialization of cipher context */
ret = EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);
...
/* Update step(s) */
ret = EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
...
/* Final step */
ret = EVP_EncryptFinal_ex(ctx, out_buf, &out_len);
...
/* Missing initialization and update */
...
/* Second final step */
ret = EVP_EncryptFinal_ex(ctx, out_buf, &out_len);
```

- You perform a cleanup of the cipher context and then perform a final step.

```
/* Initialization of cipher context */
ret = EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);
...
/* Update step(s) */
ret = EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
...
/* Cleanup of cipher context */
EVP_CIPHER_CTX_cleanup(ctx);
...
/* Second final step */
ret = EVP_EncryptFinal_ex(ctx, out_buf, &out_len);
```

Risk

Block ciphers break your data into blocks of fixed size. During encryption or decryption, the update step encrypts or decrypts your data in blocks. Any leftover data is encrypted or decrypted by the final

step. The final step adds padding to the leftover data so that it occupies one block, and then encrypts or decrypts the padded data.

If you perform the final step before performing the update steps, or perform the final step when there is no data to process, the behavior is undefined. You can also encounter run-time errors.

Fix

Perform encryption or decryption in this sequence:

- Initialization of cipher context
- Update steps
- Final step
- Cleanup of context

Examples

Missing Update Steps for Encryption Before Final Step

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

unsigned char *out_buf;
int out_len;
unsigned char key[SIZE16];
unsigned char iv[SIZE16];

void func(void) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);

    /* Initialization of cipher context */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);

    /* Missing update steps for encryption */

    /* Final encryption step */
    EVP_EncryptFinal_ex(ctx, out_buf, &out_len);
}
```

In this example, after the cipher context is initialized, there are no update steps for encrypting the data. The update steps are supposed to encrypt one or more blocks of data, leaving the final step to encrypt data that is left over in a partial block. If you perform the final step without previous update steps, the behavior is undefined.

Correction — Perform Update Steps for Encryption Before Final Step

Perform update steps for encryption before the final step. In the corrected code below, the routine `EVP_EncryptUpdate` performs the update steps.

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

unsigned char *out_buf;
int out_len;
unsigned char key[SIZE16];
unsigned char iv[SIZE16];

void func(unsigned char *src, int len) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);

    /* Initialization of cipher context */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);

    /* Update steps for encryption */
    EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);

    /* Final encryption step */
    EVP_EncryptFinal_ex(ctx, out_buf, &out_len);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_CIPHER_NO_DATA`
Impact: Medium
CWE ID: 311, 325, 372, 664

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Missing cipher final step

You do not perform a final step after update steps for encrypting or decrypting data

Description

This defect occurs when you do not perform a final step after your update steps for encrypting or decrypting data.

For instance, you do the following:

```
/* Initialization of cipher context */
ret = EVP_EncryptInit_ex(&ctx, EVP_aes_128_cbc(), NULL, key, iv);
...
/* Update step */
ret = EVP_EncryptUpdate(&ctx, out_buf, &out_len, src, len);
...
/* Missing final step */
...
/* Cleanup of cipher context */
EVP_CIPHER_CTX_cleanup(ctx);
```

Risk

Block ciphers break your data into blocks of fixed size. During encryption or decryption, the update step encrypts or decrypts your data in blocks. Any leftover data is encrypted or decrypted by the final step. The final step adds padding to the leftover data so that it occupies one block, and then encrypts or decrypts the padded data.

If you do not perform the final step, leftover data remaining in a partial block is not encrypted or decrypted. You can face incomplete or unexpected output.

Fix

After your update steps for encryption or decryption, perform a final step to encrypt or decrypt leftover data.

```
/* Initialization of cipher context */
ret = EVP_EncryptInit_ex(&ctx, EVP_aes_128_cbc(), NULL, key, iv);
...
/* Update step(s) */
ret = EVP_EncryptUpdate(&ctx, out_buf, &out_len, src, len);
...
/* Final step */
ret = EVP_EncryptFinal_ex(&ctx, out_buf, &out_len);
...
/* Cleanup of cipher context */
EVP_CIPHER_CTX_cleanup(ctx);
```

Examples

Cleanup of Cipher Context Before Final Step

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

unsigned char *out_buf;
int out_len;
unsigned char key[SIZE16];
unsigned char iv[SIZE16];

void func(unsigned char *src, int len) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);

    /* Initialization of cipher context */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);

    /* Update steps for encryption */
    EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);

    /* Missing final encryption step */

    /* Cleanup of cipher context */
    EVP_CIPHER_CTX_cleanup(ctx);
}
```

In this example, the cipher context `ctx` is cleaned up before a final encryption step. The final step is supposed to encrypt leftover data. Without the final step, the encryption is incomplete.

Correction — Perform Final Encryption Step

After your update steps for encryption, perform a final encryption step to encrypt leftover data. In the corrected code below, the routine `EVP_EncryptFinal_ex` is used to perform this final step.

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

unsigned char *out_buf;
int out_len;
unsigned char key[SIZE16];
unsigned char iv[SIZE16];

void func(unsigned char *src, int len) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);

    /* Initialization of cipher context */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);

    /* Update steps for encryption */
    EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);

    /* Final encryption step */
    EVP_EncryptFinal_ex(ctx, out_buf, &out_len);

    /* Cleanup of cipher context */
```

```
    EVP_CIPHER_CTX_cleanup(ctx);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_CIPHER_NO_FINAL`
Impact: Medium
CWE ID: 311, 325, 372, 664

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Missing cipher key

Context used for encryption or decryption is associated with NULL key or not associated with a key

Description

This defect occurs when you encrypt or decrypt data using a NULL encryption or decryption key.

Note You can initialize your cipher context with a NULL key. However, before you encrypt or decrypt your data, you must associate the cipher context with a non-NULL key.

Risk

Encryption or decryption with a NULL key can lead to run-time errors or at least, non-secure ciphertext.

Fix

Before your encryption or decryption steps

```
ret = EVP_EncryptUpdate(&ctx, out_buf, &out_len, src, len)
```

associate your cipher context `ctx` with a non-NULL key.

```
ret = EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv)
```

Sometimes, you initialize your cipher context with a non-NULL key

```
ret = EVP_EncryptInit_ex(&ctx, cipher_algo_1, NULL, key, iv)
```

but change the cipher algorithm later. When you change the cipher algorithm, you use a NULL key.

```
ret = EVP_EncryptInit_ex(&ctx, cipher_algo_2, NULL, NULL, NULL)
```

The second statement reinitializes the cipher context completely but with a NULL key. To avoid this issue, every time you initialize a cipher context with an algorithm, associate it with a key.

Examples

NULL Key Used for Encryption

```
#include <openssl/evp.h>
#include <stdlib.h>
#define fatal_error() abort()

unsigned char *out_buf;
int out_len;

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv, unsigned char *src, int len){
    if (iv == NULL)
        fatal_error();
```

```
    /* Fourth argument is cipher key */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, NULL, iv);

    /* Update step with NULL key */
    return EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
}
```

In this example, the cipher key associated with the context `ctx` is NULL. When you use this context to encrypt your data, you can encounter run-time errors.

Correction — Use Random Cipher Key

Use a strong random number generator to produce the cipher key. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define fatal_error() abort()
#define SIZE16 16

unsigned char *out_buf;
int out_len;

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv, unsigned char *src, int len){
    if (iv == NULL)
        fatal_error();
    unsigned char key[SIZE16];
    RAND_bytes(key, 16);

    /* Fourth argument is cipher key */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);

    /* Update step with non-NULL cipher key */
    return EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_CIPHER_NO_KEY`
Impact: Medium
CWE ID: 310, 320, 573, 664

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Missing data for encryption, decryption or signing operation

Data provided for public key cryptography operation is NULL or data length is zero

Description

This defect occurs when the data provided for an encryption, decryption, signing, or authentication operation is NULL or the data length is zero.

For instance, you unintentionally provide a NULL value for `in` or a zero value for `in_len` in this decryption operation:

```
ret = EVP_PKEY_decrypt(ctx, out, &out_len, in, in_len);
```

Or, you provide a NULL value for `md` or `sig`, or a zero value for `md_len` or `sig_len` in this verification operation:

```
ret = EVP_PKEY_verify(ctx, md, mdlen, sig, siglen);
```

Risk

With NULL data or zero length, the operation does not occur. The redundant operation often indicates a coding error.

Fix

Check the placement of the encryption, decryption, or signing operation. If the operation is intended to happen, make sure that the data provided is non-NULL. Set the data length to a nonzero value.

Examples

Zero Data Length for Signing Operation

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY_CTX * ctx){
  if (ctx == NULL) fatal_error();
  unsigned char* sig = (unsigned char*) "0123456789";
  unsigned char* md = (unsigned char*) "0123456789";

  ret = EVP_PKEY_verify_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_signature_md(ctx, EVP_sha256());
  if (ret <= 0) fatal_error();
  return EVP_PKEY_verify(ctx, sig, 0, md, 0);
}
```

In this example, the data lengths (third and fifth arguments to `EVP_PKEY_verify`) are zero. The operation fails.

Correction — Use Nonzero Data Length

One possible correction is to use a nonzero length for the signature and the data believed to be signed.

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY_CTX * ctx){
  if (ctx == NULL) fatal_error();
  unsigned char* sig = (unsigned char*) "0123456789";
  unsigned char* md = (unsigned char*) "0123456789";

  ret = EVP_PKEY_verify_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_signature_md(ctx, EVP_sha256());
  if (ret <= 0) fatal_error();
  return EVP_PKEY_verify(ctx, sig, 10, md, 10);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_PKEY_NO_DATA`
Impact: Medium
CWE ID: 310, 325, 372, 573

See Also

`Context initialized incorrectly for cryptographic operation` | `Find defects (-checkers)` | `Incorrect key for cryptographic algorithm` | `Missing parameters for key generation` | `Missing peer key` | `Missing private key` | `Missing public key` | `Nonsecure parameters for key generation`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Missing final step after hashing update operation

Hash is incomplete or non-secure

Description

The defect occurs when, after an update operation on a message digest context, you do not perform a final step before you clean up or reinitialize the context.

When you use message digest functions, you typically initialize a message digest context and perform at least one update step to add data into the context. You then sign, verify, or retrieve the data in the context as a final step.

Risk

A missing final step might indicate that the hash is incomplete or is non-secure.

Fix

Perform a final step to sign, verify, or retrieve date from the message digest context before you clean up or reinitialize the context.

Examples

Missing Final Step Before Context Cleanup

```
#include <stdlib.h>
#include <openssl/evp.h>


void func(unsigned char* src, int len, EVP_PKEY* pkey)
{
    int ret;

    EVP_MD_CTX ctx;
    EVP_MD_CTX_init(&ctx);

    ret = EVP_DigestVerifyInit(&ctx, NULL, EVP_sha256(), NULL, pkey);
    if (ret != 1) handle_error();

    ret = EVP_DigestVerifyUpdate(&ctx, src, len);
    if (ret != 1) handle_error();

    EVP_MD_CTX_cleanup(&ctx);
}
```

In this example, a verification context `ctx` is initialized and updated with data. The context is then cleaned up without being verified in a final step. Typically, you create a verification context to validate a previously signed message. Without the final step the signature on the message cannot be validated.

Correction — Perform Final Step Before Context Cleanup

One possible correction is to perform a final step to verify the signature of the verification context before you clean up the context.

```
#include <stdlib.h>
#include <openssl/evp.h>

unsigned char out_buf[EVP_MAX_MD_SIZE];
unsigned int out_len;

void handle_error()
{
    exit(-1);
}


void func(unsigned char* src, int len, EVP_PKEY* pkey)
{
    int ret;

    EVP_MD_CTX ctx;
    EVP_MD_CTX_init(&ctx);

    ret = EVP_DigestVerifyInit(&ctx, NULL, EVP_sha256(), NULL, pkey);
    if (ret != 1) handle_error();

    ret = EVP_DigestVerifyUpdate(&ctx, src, len);
    if (ret != 1) handle_error();

    ret = EVP_DigestVerifyFinal(&ctx, out_buf, out_len);
    if (ret != 1) handle_error();

    EVP_MD_CTX_cleanup(&ctx);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_MD_NO_FINAL`
Impact: Medium
CWE ID: 573

See Also

`Find defects (-checkers)` | `No data added into context` | `Nonsecure hash algorithm`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Missing hash algorithm

Context in EVP routine is initialized without a hash algorithm

Description

This defect occurs when you use a message digest context in these EVP routines, but you initialize the context without specifying a hash algorithm.

- `EVP_DigestFinal`
- `EVP_DigestSignFinal`
- `EVP_SignFinal`
- `EVP_VerifyFinal`

Risk

Using a message digest context that was initialized without an algorithm to perform a hashing operation might result in a run-time error. Even if the hashing operation is successful, the resulting digest is not secure.

Fix

Specify a hash algorithm when you initial a message digest context that you use in an EVP routine.

Examples

Context Used in EVP Routine After Context Cleanup

```
#include <openssl/evp.h>

void func(unsigned char* src, int len)
{
    EVP_MD_CTX ctx;
    EVP_MD_CTX_init(&ctx);

    EVP_VerifyInit(&ctx, EVP_sha256());
    EVP_MD_CTX_cleanup(&ctx);
    EVP_VerifyUpdate(&ctx, src, len);
}
```

In this example, context `ctx` is initialized with secure hash algorithm SHA-256. But, `ctx` is cleaned up before it is used by `EVP_VerifyUpdate`. The clean up of `ctx` frees up its resources and reinitializes it without a hash algorithm. The hashing operation of `EVP_VerifyUpdate` might result in a run-time error.

Correction — Clean Up Context Only After You No Longer Need It

One possible correction is to clean up the digest context only after you no longer need it.

```
#include <openssl/evp.h>

void func(unsigned char* src, int len)
{
    EVP_MD_CTX ctx;
    EVP_MD_CTX_init(&ctx);

    EVP_VerifyInit(&ctx, EVP_sha256());
    EVP_VerifyUpdate(&ctx, src, len);
    EVP_MD_CTX_cleanup(&ctx);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: CRYPTO_MD_NO_ALGORITHM
Impact: Medium
CWE ID: 573

See Also

`Find defects (-checkers)` | `Nonsecure hash algorithm`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019b

Missing padding for RSA algorithm

Context used in encryption or signing operation is not associated with any padding

Description

This defect occurs when you perform RSA encryption or signature by using a context object without associating the object with a padding scheme.

For instance, you perform encryption by using a context object that was initially not associated with a specific padding.

```
ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_NO_PADDING);
...
ret = EVP_PKEY_encrypt(ctx, out, &out_len, in, in_len)
```

Risk

Padding schemes remove determinism from the RSA algorithm and protect RSA operations from certain kinds of attack. Padding ensures that a given message does not lead to the same ciphertext each time it is encrypted. Without padding, an attacker can launch chosen-plaintext attacks against the cryptosystem.

Fix

Before performing an RSA operation, associate the context object with a padding scheme that is compatible with the operation.

- Encryption: Use the OAEP padding scheme.

  For instance, use the `EVP_PKEY_CTX_set_rsa_padding` function with the argument `RSA_PKCS1_OAEP_PADDING` or the `RSA_padding_add_PKCS1_OAEP` function.

  ```
  ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_PADDING);
  ```

  You can also use the PKCS#1v1.5 or SSLv23 schemes. Be aware that these schemes are considered insecure.

  You can then use functions such as `EVP_PKEY_encrypt` / `EVP_PKEY_decrypt` or `RSA_public_encrypt` / `RSA_private_decrypt` on the context.
- Signature: Use the RSA-PSS padding scheme.

  For instance, use the `EVP_PKEY_CTX_set_rsa_padding` function with the argument `RSA_PKCS1_PSS_PADDING`.

  ```
  ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_PSS_PADDING);
  ```

  You can also use the ANSI X9.31, PKCS#1v1.5, or SSLv23 schemes. Be aware that these schemes are considered insecure.

  You can then use functions such as the `EVP_PKEY_sign`-`EVP_PKEY_verify` pair or the `RSA_private_encrypt`-`RSA_public_decrypt` pair on the context.

If you perform two kinds of operation with the same context, after the first operation, reset the padding scheme in the context before the second operation.

Examples

Encryption Without Padding

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
size_t out_len;


int func(unsigned char *src, size_t len){
  EVP_PKEY_CTX *ctx;
  EVP_PKEY* pkey;

  /* Key generation */
  ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA,NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 2048);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_keygen(ctx, &pkey);
  if (ret <= 0) fatal_error();

  /* Encryption */
  EVP_PKEY_CTX_free(ctx);
  ctx = EVP_PKEY_CTX_new(pkey,NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_encrypt_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_encrypt(ctx, out_buf, &out_len, src, len);
}
```

In this example, before encryption with `EVP_PKEY_encrypt`, a specific padding is not associated with the context object `ctx`.

Correction — Set Padding in Context Before Encryption

One possible correction is to set the OAEP padding scheme in the context.

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)
```

```
int ret;
unsigned char *out_buf;
size_t out_len;


int func(unsigned char *src, size_t len){
  EVP_PKEY_CTX *ctx;
  EVP_PKEY* pkey;

  /* Key generation */
  ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA,NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 2048);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_keygen(ctx, &pkey);
  if (ret <= 0) fatal_error();

  /* Encryption */
  EVP_PKEY_CTX_free(ctx);
  ctx = EVP_PKEY_CTX_new(pkey,NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_encrypt_init(ctx);
  ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_PADDING);
  if (ret <= 0) fatal_error();
  if (ret <= 0) fatal_error();
  return EVP_PKEY_encrypt(ctx, out_buf, &out_len, src, len);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_RSA_NO_PADDING`
Impact: Medium
CWE ID: 310, 326, 327, 780

See Also

`Find defects (-checkers)` | `Incompatible padding for RSA algorithm operation` | `Missing blinding for RSA algorithm` | `Nonsecure RSA public exponent` | `Weak padding for RSA algorithm`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Missing parameters for key generation

Context used for key generation is associated with NULL parameters

Description

This defect occurs when you perform a key generation step with a context object without first associating the object with required parameters.

For instance, you associate a `EVP_PKEY_CTX` context object with an empty `EVP_PKEY` object `params` before key generation :

```
EVP_PKEY * params = EVP_PKEY_new();
...
EVP_PKEY_CTX * ctx = EVP_PKEY_CTX_new(params, NULL);
...
EVP_PKEY_keygen(ctx, &pkey);
```

Risk

Without appropriate parameters, the key generation step does not occur. The redundant operation often indicates a coding error.

Fix

Check the placement of the key generation step. If the operation is intended, make sure that the parameters are set before key generation.

Certain algorithms use default parameters. For instance, if you specify the DSA algorithm when creating the `EVP_PKEY_CTX` object, a default key length of 1024 bits is used:

```
kctx = EVP_PKEY_CTX_new_id(EVP_PKEY_DSA, NULL);
```

Specifying the algorithm during context creation is sufficient to avoid this defect. Only if you use the Elliptic Curve (EC) algorithm, you must also specify the curve explicitly before key generation.

However, the default parameters can generate keys that are too weak for encryption. Weak parameters can trigger another defect. To change default parameters, use functions specific to the algorithm. For instance, to set parameters, you can use these functions:

- Diffie-Hellman (DH): Use `EVP_PKEY_CTX_set_dh_paramgen_prime_len` and `EVP_PKEY_CTX_set_dh_paramgen_generator`.
- Digital Signature Algorithm (DSA): Use `EVP_PKEY_CTX_set_dsa_paramgen_bits`.
- RSA: Use `EVP_PKEY_CTX_set_rsa_padding`, `EVP_PKEY_CTX_set_rsa_pss_saltlen`, `EVP_PKEY_CTX_set_rsa_rsa_keygen_bits`, and `EVP_PKEY_CTX_set_rsa_keygen_pubexp`.
- Elliptic curve (EC): Use `EVP_PKEY_CTX_set_ec_paramgen_curve_nid` and `EVP_PKEY_CTX_set_ec_param_enc`.

Examples

Empty Parameters During Key Generation

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY *pkey){
  EVP_PKEY * params = EVP_PKEY_new();
  if (params == NULL) fatal_error();

  EVP_PKEY_CTX * ctx = EVP_PKEY_CTX_new(params, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_keygen(ctx, &pkey);
}
```

In this example, the context object `ctx` is associated with an empty parameter object `params`. The context object does not have the required parameters for key generation.

Correction — Specify Algorithm During Context Creation

One possible correction is to specify an algorithm, such as RSA, during context creation. For stronger encryption, use 2048 bits for key length instead of the default 1024 bits.

```
#include <openssl/evp.h>
#include <openssl/rsa.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY *pkey){
  EVP_PKEY_CTX * ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();

  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 2048);
  if (ret <= 0) fatal_error();

  return EVP_PKEY_keygen(ctx, &pkey);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_PKEY_NO_PARAMS`
Impact: Medium

CWE ID: 310, 325, 372, 573

See Also

`Context initialized incorrectly for cryptographic operation` | `Find defects (-checkers)` | `Incorrect key for cryptographic algorithm` | `Missing data for encryption, decryption or signing` | `Missing peer key` | `Missing private key` | `Missing public key` | `Nonsecure parameters for key generation`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Missing peer key

Context used for shared secret derivation is associated with NULL peer key or not associated with a peer key at all

Description

This defect occurs when you use a context object for shared secret derivation but you have not previously associated the object with a non-NULL peer key.

For instance, you initialize the context object, and then use the object for shared secret derivation without an intermediate step where the object is associated with a peer key:

```
EVP_PKEY_derive_init(ctx);
/* Missing step for associating peer key with context */
ret = EVP_PKEY_derive(ctx, out_buf, &out_len);
```

The counterpart checker `Missing private key` checks for a private key in shared secret derivation.

Risk

Without a peer key, the shared secret derivation step does not occur. The redundant operation often indicates a coding error.

Fix

Check the placement of the shared secret derivation step. If the operation is intended, make sure that you have completed these steps prior to the operation:

- Generate a non-NULL peer key.

  For instance:

  ```
  EVP_PKEY* peerkey = NULL;
  EVP_PKEY_keygen(EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL), &peerkey);
  ```

- Associate a non-NULL context object with the peer key.

  For instance:

  ```
  EVP_PKEY_derive_set_peer(ctx,peerkey);
  ```

Examples

Missing Step for Associating Peer Key with Context

```
#include <stddef.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
```

```
unsigned char *out_buf;
size_t out_len;

int func(EVP_PKEY *pkey){
  if (pkey == NULL) fatal_error();

  EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new(pkey, NULL);
  if (ctx == NULL) fatal_error();
  ret = EVP_PKEY_derive_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_derive(ctx, out_buf, &out_len);
}
```

In this example, the context object `ctx` is associated with a private key but not a peer key. The `EVP_PKEY_derive` function uses this context object for shared secret derivation.

Correction — Set Peer Key in Context

One possible correction is to use the function `EVP_PKEY_derive_set_peer` and associate a peer key with the context object. Make sure that the peer key is non-NULL.

```
#include <stddef.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
size_t out_len;

int func(EVP_PKEY *pkey,  EVP_PKEY* peerkey){
  if (pkey == NULL) fatal_error();
  if (peerkey == NULL) fatal_error();

  EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new(pkey, NULL);
  if (ctx == NULL) fatal_error();
  ret = EVP_PKEY_derive_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_derive_set_peer(ctx,peerkey);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_derive(ctx, out_buf, &out_len);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_PKEY_NO_PEER`
Impact: Medium
CWE ID: 310, 320, 573, 664

See Also

`Context initialized incorrectly for cryptographic operation` | `Find defects (-checkers)` | `Incorrect key for cryptographic algorithm` | `Missing parameters for`

```
key generation|Missing data for encryption, decryption or signing|Missing
private key|Missing public key|Nonsecure parameters for key generation
```

Topics
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Missing private key

Context used for cryptography operation is associated with NULL private key or not associated with a private key at all

Description

This defect occurs when you use a context object for decryption, signature, or shared secret derivation but you have not previously associated the object with a non-NULL private key.

For instance, you initialize the context object with a NULL private key and use the object for decryption later.

```
ctx = EVP_PKEY_CTX_new(pkey, NULL);
...
ret = EVP_PKEY_decrypt_init(ctx);
...
ret = EVP_PKEY_decrypt(ctx, out, &out_len, in, in_len);
```

The counterpart checker `Missing public key` checks for a public key in encryption and authentication operations. The checker `Missing peer key` checks for a peer key in shared secret derivation.

Risk

Without a private key, the decryption, signature, or shared secret derivation step does not occur. The redundant operation often indicates a coding error.

Fix

Check the placement of the operation (decryption, signature, or shared secret derivation). If the operation is intended, make sure you have completed these steps prior to the operation:

- Generate a non-NULL private key.

  For instance:

  ```
  EVP_PKEY *pkey = NULL;
  kctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);

  EVP_PKEY_keygen_init(kctx);
  EVP_PKEY_CTX_set_rsa_keygen_bits(kctx, RSA_2048BITS);
  EVP_PKEY_keygen(kctx, &pkey);
  ```

- Associate a non-NULL context object with the private key.

  For instance:

  ```
  ctx = EVP_PKEY_CTX_new(pkey, NULL);
  ```

  Note: If you use `EVP_PKEY_CTX_new_id` instead of `EVP_PKEY_CTX_new`, you are not associating the context object with a private key.

Examples

Missing Step for Associating Private Key with Context

```
#include <stddef.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
size_t out_len;

int func(unsigned char *src, size_t len){
  EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_decrypt_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_decrypt(ctx, out_buf, &out_len, src, len);
}
```

In this example, the context object `ctx` is initialized with `EVP_PKEY_CTX_new_id` instead of `EVP_PKEY_CTX_new`. The function `EVP_PKEY_CTX_new_id` does not associate the context object with a key. However, the `EVP_PKEY_decrypt` function uses this object for decryption.

Correction — Associate Private Key with Context During Initialization

One possible correction is to use the `EVP_PKEY_CTX_new` function for context initialization and associate a private key with the context object. In the following correction, the private key `pkey` is obtained from an external source and checked for NULL before use.

```
#include <stddef.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
size_t out_len;

int func(unsigned char *src, size_t len, EVP_PKEY *pkey){
  if (pkey == NULL) fatal_error();

  EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new(pkey, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_decrypt_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_decrypt(ctx, out_buf, &out_len, src, len);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_PKEY_NO_PRIVATE_KEY`
Impact: Medium
CWE ID: 310, 320, 573, 664

See Also

`Context initialized incorrectly for cryptographic operation` | `Find defects (-checkers)` | `Incorrect key for cryptographic algorithm` | `Missing parameters for key generation` | `Missing data for encryption, decryption or signing` | `Missing peer key` | `Missing public key` | `Nonsecure parameters for key generation`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Missing private key for X.509 certificate

Missing key might result in run-time error or non-secure encryption

Description

The defect occurs when you load a X.509 certificate file into the SSL context but you do not load the corresponding private key, or the key that you load into the context is null.

Typically, in a TLS/SSL exchange, the server proves its identity during a TLS/SSL handshake by sending a X.509 certificate that contains information about the server and a public key. The client that receives the certificate uses the public key to encrypt and send a pre-master secret that can only be decrypted with the corresponding private key. The server uses the decrypted pre-master secret and other exchanged messages to generate session keys that are used to encrypt the communication session.

The checker raises no defect if:

- You pass the SSL context as an argument to the function that calls `SSL_new`.
- You declare the SSL context outside the scope of the function handling the connection.

Risk

Not loading the private key for a X.509 certificate might result in a run-time error on non-secure encryption.

Fix

Load the private key of the X.509 certificate into the SSL context by calling `SSL_CTX_use_PrivateKey_file` or load the private key into the SSL structure by calling `SSL_use_PrivateKey_file`.

Examples

No Private Key Loaded Into SSL Context

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>
#define SSL_SERVER_CRT "server.pem"

#define fatal_error() exit(-1)

void load_cert(SSL_CTX* ctx, const char* certfile)
{
    int ret = SSL_CTX_use_certificate_file(ctx, certfile, SSL_FILETYPE_PEM);
    if (ret <= 0) fatal_error();
}

void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;

    /* creation context for the SSL protocol */
    ctx = SSL_CTX_new(SSLv23_server_method());
    if (ctx == NULL) fatal_error();

    /* context configuration */
```

```
    load_cert(ctx, SSL_SERVER_CRT);

    /* Handle connection */
    ssl = SSL_new(ctx);
    ret = SSL_accept(ssl);
    if (ret <= 0) fatal_error();

    SSL_free(ssl);
    SSL_CTX_free(ctx);
}
```

In this example, SSL context `ctx` is initiated with server role and the function `load_cert` loads the server certificate into `ctx`. The server then waits for a client to initiate a handshake. However, since the private key is not loaded into the SSL structure, the server cannot decrypt the pre-master secret that a client sends, and the handshake fails.

Correction — Load Private Key into SSL Context

One possible correction is to load the private key into the SSL context after you load the server certificate file.

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>
#define SSL_SERVER_CRT "server.pem"
#define SSL_SERVER_KEY "server.key"

#define fatal_error() exit(-1)

void load_cert(SSL_CTX* ctx, const char* certfile)
{
    int ret = SSL_CTX_use_certificate_file(ctx, certfile, SSL_FILETYPE_PEM);
    if (ret <= 0) fatal_error();

    ret = SSL_CTX_use_PrivateKey_file(ctx, SSL_SERVER_KEY, SSL_FILETYPE_PEM);
    if (ret <= 0) fatal_error();
}

void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;

    /* creation context for the SSL protocol */
    ctx = SSL_CTX_new(SSLv23_server_method());
    if (ctx == NULL) fatal_error();

    /* context configuration */
    load_cert(ctx, SSL_SERVER_CRT);

    /* Handle connection */
    ssl = SSL_new(ctx);
    ret = SSL_accept(ssl);
    if (ret <= 0) fatal_error();

    SSL_free(ssl);
    SSL_CTX_free(ctx);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_SSL_NO_PRIVATE_KEY`
Impact: Medium
CWE ID: 573

See Also

`Find defects (-checkers)|Missing X.509 certificate|Missing certification authority list`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Missing public key

Context used for cryptography operation is associated with NULL public key or not associated with a public key at all

Description

This defect occurs when you use a context object for encryption or signature authentication but you have not previously associated the object with a non-NULL public key.

For instance, you initialize the context object with a NULL public key and use the object for encryption later.

```
ctx = EVP_PKEY_CTX_new(pkey, NULL);
...
ret = EVP_PKEY_encrypt_init(ctx);
...
ret = EVP_PKEY_encrypt(ctx, out, &out_len, in, in_len);
```

The counterpart checker `Missing private key` checks for a private key in decryption and signature operations.

Risk

Without a public key, the encryption or signature authentication step does not happen. The redundant operation often indicates a coding error.

Fix

Check the placement of the operation (encryption or signature authentication). If the operation is intended to happen, make sure you have done these steps prior to the operation:

- You generated a non-NULL public key.

  For instance:

  ```
  EVP_PKEY *pkey = NULL;
  kctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);

  EVP_PKEY_keygen_init(kctx);
  EVP_PKEY_CTX_set_rsa_keygen_bits(kctx, RSA_2048BITS);
  EVP_PKEY_keygen(kctx, &pkey);
  ```

- You associated a non-NULL context object with the public key.

  For instance:

  ```
  ctx = EVP_PKEY_CTX_new(pkey, NULL);
  ```

  Note: If you use `EVP_PKEY_CTX_new_id` instead of `EVP_PKEY_CTX_new`, you are not associating the context object with a public key.

Examples

Missing Step for Associating Private Key with Context

```
#include <stddef.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
size_t out_len;

int func(unsigned char *src, size_t len){
  EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_encrypt_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_encrypt(ctx, out_buf, &out_len, src, len);
}
```

In this example, the context object `ctx` is initialized with `EVP_PKEY_CTX_new_id` instead of `EVP_PKEY_CTX_new`. The function `EVP_PKEY_CTX_new_id` does not associate the context object with a key. However, the `EVP_PKEY_encrypt` function uses this object for decryption.

Correction — Associate Public Key with Context During Initialization

One possible correction is to use the `EVP_PKEY_CTX_new` function for context initialization and associate a public key with the context object. In the following correction, the public key `pkey` is obtained from an external source and checked for NULL before use.

```
#include <stddef.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
size_t out_len;

int func(unsigned char *src, size_t len, EVP_PKEY *pkey){
  if (pkey == NULL) fatal_error();

  EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new(pkey, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_encrypt_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_encrypt(ctx, out_buf, &out_len, src, len);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_PKEY_NO_PUBLIC_KEY`
Impact: Medium
CWE ID: 310, 320, 573, 664

See Also

`Context initialized incorrectly for cryptographic operation` | `Find defects (-checkers)` | `Incorrect key for cryptographic algorithm` | `Missing parameters for key generation` | `Missing data for encryption, decryption or signing` | `Missing peer key` | `Missing private key` | `Nonsecure parameters for key generation`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Missing salt for hashing operation

Hashed data is vulnerable to rainbow table attack

Description

This defect occurs when you use a digest context in these functions, but you hash data into the context only once or you use a null salt in all subsequent hashing operations. A salt is random data that you use to improve the security of a hashing operation. The hashing operation takes the salt as an input to produce a more secure hashed value.

- `EVP_DigestFinal`
- `EVP_DigestSignUpdate`
- `EVP_DigestVerifyUpdate`
- `SHA*_Final` family of functions

Missing salt for hashing operation raises no defect if no information is available about the context. For instance, if the context is passed as an argument to the function that calls the hashing operation or if the context is declared outside the scope of the function. For example, no defect is raised in this code snippet.

```
EVP_MD_CTX ctx_global;

void foo(EVP_MD_CTX* ctx) {
//ctx passed as argument of func()
    EVP_DigestFinal(ctx, out_buf, &out_len); //no defect
}

void bar() {
// ctx_global declared outside of bar()
    EVP_DigestFinal(&ctx_glob, out_buf, &out_len); //no defect
}
```

Risk

Hashing the same data without a salt results in the same hashed value. For instance, if you hash user passwords and two users have the same passwords, the hashed passwords are identical. The hashing is then vulnerable to precomputed rainbow attacks.

Fix

Provide a salt when you hash data.

Examples

Data Hashed Into Context Only Once

```
#include <openssl/evp.h>
#include <cstring>

unsigned char* out_buf;
```

```
unsigned int out_len;

void func()
{
    const char* src = "toto";
    EVP_MD_CTX ctx;

    EVP_DigestInit(&ctx, EVP_sha256());
    EVP_DigestUpdate(&ctx, src, strlen(src));
    EVP_DigestFinal(&ctx, out_buf, &out_len);
    EVP_cleanup();
}
```

In this example, context `ctx` is initialized with secure hashing algorithm SHA-256, then `EVP_DigestUpdate` hashes `src` into `ctx`. Because `EVP_DigestUpdate` is called only once, no salt can be provided to improve the security of the hashing operation. The digest value that `EVP_DigestFinal` retrieves is then vulnerable to precomputed rainbow attacks.

Correction — Hash Salt Into Context After Initial Data Hash

One possible correction is to hash a salt into the context `ctx` after the first hashing operation. The resulting digest value that `EVP_DigestFinal` retrieves is more secure.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <cstring>

#define BUFF_SIZE_32 32

unsigned char* out_buf;
unsigned int out_len;

void func()
{
    const char* src = "toto";
    const char* salt;

    RAND_bytes((unsigned char*)salt, BUFF_SIZE_32);
    EVP_MD_CTX ctx;

    EVP_DigestInit(&ctx, EVP_sha256());
    EVP_DigestUpdate(&ctx, src, strlen(src));
    EVP_DigestUpdate(&ctx, salt, BUFF_SIZE_32);
    EVP_DigestFinal(&ctx, out_buf, &out_len);
    EVP_cleanup();
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_MD_NO_SALT`
Impact: Medium
CWE ID: 759

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019b

Missing X.509 certificate

Server or client cannot be authenticated

Description

This defect occurs when you use a context to handle TLS/SSL connections with these functions, but you do not load an X.509 certificate into the context.

- `SSL_accept`
- `SSL_connect`
- `SSL_do_handshake`
- `SSL_write`
- `SSL_read`
- `BIO_do_accept`
- `BIO_do_connect`
- `BIO_do_handshake`

An X.509 certificate is a digital certificate that is issued to an entity. It contains information that identifies the entity. The certificate is used to authenticate connections to the entity identified in the certificate.

The checker raises a defect if:

- For a server authentication, no certificate is loaded before handling a connection.
- For a client authentication, the client attempts to connect to a server a second time after getting an `SSL_ERROR_WANT_X509_LOOKUP` error on the first connection attempt.

Risk

When you do not load an X.509 certificate into the context to handle TLS/SSL connections, the connection is not secure and is vulnerable to man-in-the-middle (MITM) attacks.

Fix

Load an X.509 certificate into the context you create to handle TLS/SSL connections.

Examples

SSL Structure Created From Context with Missing Certificate

```
#include <openssl/ssl.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <openssl/err.h>

unsigned char* buf;
int len;

SSL_CTX* InitServerCTX(void)
{
    SSL_CTX* ctx;
```

```
    OpenSSL_add_all_algorithms();
    ctx = SSL_CTX_new(SSLv23_server_method());
    SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2 | SSL_OP_NO_SSLv3 | SSL_OP_NO_TLSv1);
    if (ctx == NULL) {
        /*handle errors */
    }
    return ctx;
}

int OpenListener(int port)
{
    /* Create server socket */
}

void func()
{
    SSL_CTX* ctx;
    int server, port;
    int ret;
    SSL_library_init();

    ctx = InitServerCTX();
    server = OpenListener(port);
    while (1) {
        struct sockaddr_in addr;
        socklen_t len = sizeof(addr);
        SSL* ssl;

        int client = accept(server, (struct sockaddr*)&addr, &len);
        printf("Connection: %s:%d\n", inet_ntoa(addr.sin_addr), ntohs(addr.sin_port));
        ssl = SSL_new(ctx);
        SSL_set_fd(ssl, client);
        ret = SSL_accept(ssl);
        if (SSL_get_error(ssl, ret) <= 0)
            /* Serve connection */;
        else
            SSL_free(ssl);
    }
    close(server);
    SSL_CTX_free(ctx);
}
```

In this example, `InitServerCTX()` initializes context `ctx` for TLS/SSL connections, but no certificate is loaded into `ctx`. When `SSL_accept` checks the TLS/SLL handshake for the SSL structure created from `ctx`, there is no certificate available to authenticate the server.

Correction — Before Creating a SSL Structure, Load Certificate Into Context

One possible correction is to, before you create a SSL structure, load a certificate into the context you create for TLS/SSL connections, for instance with `SSL_CTX_use_certificate_file`.

```
#include <openssl/ssl.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <openssl/err.h>

unsigned char* buf;
int len;

SSL_CTX* InitServerCTX(void)
{
    SSL_CTX* ctx;
    OpenSSL_add_all_algorithms();
    ctx = SSL_CTX_new(SSLv23_server_method());
    SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2 | SSL_OP_NO_SSLv3 | SSL_OP_NO_TLSv1);
    if (ctx == NULL) {
        /*handle errors */
    }
    return ctx;
}

void LoadCertificates(SSL_CTX* ctx, char* CertFile, char* KeyFile)
{
    if (SSL_CTX_use_certificate_file(ctx, CertFile, SSL_FILETYPE_PEM) <= 0) {
        /* Handle errors */
    }
```

```
}

int OpenListener(int port)
{
    /* Create server socket */
}

void func()
{
    SSL_CTX* ctx;
    int server, port;
    int ret;
    SSL_library_init();

    ctx = InitServerCTX();
    LoadCertificates(ctx, "mycert.pem", "mycert.pem");
    server = OpenListener(port);
    while (1) {
        struct sockaddr_in addr;
        socklen_t len = sizeof(addr);
        SSL* ssl;

        int client = accept(server, (struct sockaddr*)&addr, &len);
        printf("Connection: %s:%d\n", inet_ntoa(addr.sin_addr), ntohs(addr.sin_port));
        ssl = SSL_new(ctx);
        SSL_set_fd(ssl, client);
        ret = SSL_accept(ssl);
        if (SSL_get_error(ssl, ret) <= 0)
            /* Serve connection */;
        else
            SSL_free(ssl);
    }
    close(server);
    SSL_CTX_free(ctx);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_SSL_NO_CERTIFICATE`
Impact: Medium
CWE ID: 310

See Also

`Find defects (-checkers)|Missing certification authority list`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019b

No data added into context

Performing hash operation on empty context might cause run-time errors

Description

The defect occurs when you only update a message digest context with null data, or you perform a final step on a message digest context without performing any update step.

When you use message digest functions, you typically initialize a message digest context and perform at least one update step to add data into the context. You then sign, verify, or retrieve the data in the context as a final step.

The checker raises no defect if no information is available about the context. For instance, if the context is passed as an argument to the function that calls the hashing operation or if the context is declared outside the scope of the function. For example, no defect is raised in this code snippet.

```
void bar(unsigned char* src, int len, EVP_MD_CTX *ctx) {
    //ctx passed as argument of bar()
    EVP_DigestFinal(ctx, out_buf, &out_len); //no defect
}
EVP_MD_CTX glob_ctx;
void foo(unsigned char* src, int len) {
    //glob_ctx declared outside scope of foo()
    EVP_DigestFinal(&glob_ctx, out_buf, &out_len); //no defect
}
```

Risk

Performing an update step on a context with null data might result in a run-time error.

Performing a final step on a context with no data might result in unexpected behavior.

Fix

Perform at least one update operation with non-null data on a message digest context before you sign, verify, or retrieve the data in the context.

Examples

No Update Step Before Final Step

```
#include <openssl/evp.h>
#include <stdio.h>

unsigned char out_buf[EVP_MAX_MD_SIZE];
unsigned int out_len;

void func(unsigned char* src, int len)
{
    EVP_MD_CTX ctx;
    EVP_MD_CTX_init(&ctx);
```

```
    EVP_DigestInit(&ctx, EVP_sha256());
    EVP_DigestUpdate(&ctx, src, len);
    EVP_MD_CTX_init(&ctx);
    EVP_DigestFinal(&ctx, out_buf, &out_len);
}
```

In this example, the message digest context `ctx` is initialized and an update operation is performed to add data `src` into the context. The context is then reinitialized but no data is added to `ctx` before `EVP_DigestFinal` attempts to retrieve data from `ctx`, which results in an error.

Correction — Perform Final Step Before Reinitializing Context

One possible correction is to perform the final step that retrieves data from the context before you reinitialize the context.

```
#include <openssl/evp.h>
#include <stdio.h>

unsigned char out_buf[EVP_MAX_MD_SIZE];
unsigned int out_len;

void func(unsigned char* src, int len)
{
    EVP_MD_CTX ctx;
    EVP_MD_CTX_init(&ctx);


    EVP_DigestInit(&ctx, EVP_sha256());
    EVP_DigestUpdate(&ctx, src, len);
    EVP_DigestFinal(&ctx, out_buf, &out_len);
    EVP_MD_CTX_init(&ctx);
}
```

No Data Added to Context

```
#include <openssl/evp.h>
#include <stdio.h>

unsigned char out_buf[EVP_MAX_MD_SIZE];
unsigned int out_len;

void func(unsigned char* src, int len)
{
    EVP_MD_CTX ctx;
    EVP_MD_CTX_init(&ctx);
    size_t cnt = 0;

    EVP_DigestInit(&ctx, EVP_sha256());
    EVP_DigestUpdate(&ctx, src, cnt);
    EVP_DigestFinal(&ctx, out_buf, &out_len);
}
```

In this example, zero bytes of data is hashed into the message digest context during the update operation. Retrieving data from the context in the final step results in unexpected behavior.

Correction — Add non-Null Data Into Context

A possible correction is to add data into the context during the update step before you retrieve data from the context.

```
#include <openssl/evp.h>
#include <stdio.h>

unsigned char out_buf[EVP_MAX_MD_SIZE];
unsigned int out_len;

void func(unsigned char* src, int len)
{
    EVP_MD_CTX ctx;
    EVP_MD_CTX_init(&ctx);

    EVP_DigestInit(&ctx, EVP_sha256());
    EVP_DigestUpdate(&ctx, src, len);
    EVP_DigestFinal(&ctx, out_buf, &out_len);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_MD_NO_DATA`
Impact: Medium
CWE ID: 325

See Also

`Find defects (-checkers)|Missing final step after hashing update operation`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Nonsecure hash algorithm

Context used for message digest creation is associated with weak algorithm

Description

This defect occurs when you use a cryptographic hash function that is proven to be weak against certain forms of attack.

The hash functions flagged by this checker include SHA-0, SHA-1, MD4, MD5, and RIPEMD-160. The checker detects the use of these hash functions in:

- Functions from the EVP API such as `EVP_DigestUpdate` or `EVP_SignUpdate`.
- Functions from the low level API such as `SHA1_Update` or `MD5_Update`.

Risk

You use a hash function to create a message digest from input data and thereby ensure integrity of your data. The hash functions flagged by this checker use algorithms with known weaknesses that an attacker can exploit. The attacks can comprise the integrity of your data.

Fix

Use a more secure hash function. For instance, use the later SHA functions such as SHA-224, SHA-256, SHA-384, and SHA-512.

Examples

Use of MD5 Algorithm

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
unsigned int out_len;

void func(unsigned char *src, size_t len, EVP_PKEY* pkey){
  EVP_MD_CTX* ctx = EVP_MD_CTX_create();

  ret = EVP_SignInit_ex(ctx, EVP_md5(), NULL);
  if (ret != 1) fatal_error();

  ret = EVP_DigestUpdate(ctx,src,len);

  if (ret != 1) fatal_error();

  ret = EVP_SignFinal(ctx, out_buf, &out_len, pkey);
  if (ret != 1) fatal_error();
}
```

In this example, during initialization with `EVP_SignInit_ex`, the context object is associated with the weak hash function MD5. The checker flags the usage of this context in the update step with `EVP_DigestUpdate`.

Correction — Use SHA-2 Family Function

One possible correction is to use a hash function from the SHA-2 family, such as SHA-256.

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
unsigned int out_len;

void func(unsigned char *src, size_t len, EVP_PKEY* pkey){
  EVP_MD_CTX* ctx = EVP_MD_CTX_create();

  ret = EVP_SignInit_ex(ctx, EVP_sha256(), NULL);
  if (ret != 1) fatal_error();

  ret = EVP_SignUpdate(ctx, src, len);
  if (ret != 1) fatal_error();

  ret = EVP_SignFinal(ctx, out_buf, &out_len, pkey);
  if (ret != 1) fatal_error();
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_MD_WEAK_HASH`
Impact: Medium
CWE ID: 310, 327, 328, 353, 522

See Also

`Context initialized incorrectly for digest operation` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Nonsecure parameters for key generation

Context used for key generation is associated with weak parameters

Description

This defect occurs when you attempt key generation by using an `EVP_PKEY_CTX` context object that is associated with weak parameters. What constitutes a weak parameter depends on the public key algorithm used. In the DSA algorithm, a weak parameter can be the result of setting an insufficient parameter length.

For instance, you set the number of bits used for DSA parameter generation to 512 bits, and then use the parameters for key generation:

```
EVP_PKEY_CTX *pctx,*kctx;
EVP_PKEY *params, *pkey;

/* Initializations for parameter generation */
pctx = EVP_PKEY_CTX_new_id(EVP_PKEY_DSA, NULL);
params = EVP_PKEY_new();

/* Parameter generation */
ret = EVP_PKEY_paramgen_init(pctx);
ret = EVP_PKEY_CTX_set_dsa_paramgen_bits(pctx, KEYLEN_512BITS);
ret = EVP_PKEY_paramgen(pctx, &params);

/* Initializations for key generation */
kctx = EVP_PKEY_CTX_new(params, NULL);
pkey = EVP_PKEY_new();

/* Key generation */
ret = EVP_PKEY_keygen_init(kctx);
ret = EVP_PKEY_keygen(kctx, &pkey);
```

Risk

Weak parameters lead to keys that are not sufficiently strong for encryption and expose sensitive information to known ways of attack.

Fix

Depending on the algorithm, use these parameters:

- Diffie-Hellman (DH): Set the length of the DH prime parameter to 2048 bits.

  ```
  ret = EVP_PKEY_CTX_set_dh_paramgen_prime_len(pctx, 2048);
  ```

  Set the DH generator to 2 or 5.

  ```
  ret = EVP_PKEY_CTX_set_dh_paramgen_generator(pctx, 2);
  ```

- Digital Signature Algorithm (DSA): Set the number of bits used for DSA parameter generation to 2048 bits.

  ```
  ret = EVP_PKEY_CTX_set_dsa_paramgen_bits(pctx, 2048);
  ```

- RSA: Set the RSA key length to 2048 bits.

  ```
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(kctx, 2048);
  ```

- Elliptic curve (EC): Avoid using curves that are known to be broken, for instance, `X9_62_prime256v1`. Use, for instance, `sect239k1`.

  ```
  ret = EVP_PKEY_CTX_set_ec_paramgen_curve_nid(pctx, NID_sect239k1);
  ```

Examples

Insufficient Bits for RSA Key Generation

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY *pkey){
  EVP_PKEY_CTX * ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 512);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_keygen(ctx, &pkey);
}
```

In this example, the RSA key generation uses 512 bits, which makes the generated key vulnerable to attacks.

Correction — Use 2048 bits

Use 2048 bits for RSA key generation.

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY *pkey){
  EVP_PKEY_CTX * ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 2048);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_keygen(ctx, &pkey);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_PKEY_WEAK_PARAMS`
Impact: Medium
CWE ID: 310, 326, 327, 522

See Also

`Context initialized incorrectly for cryptographic operation` | `Find defects (-checkers)` | `Incorrect key for cryptographic algorithm` | `Missing parameters for key generation` | `Missing data for encryption, decryption or signing` | `Missing peer key` | `Missing private key` | `Missing public key`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

External Websites

https://safecurves.cr.yp.to/
https://csrc.nist.gov/publications/detail/fips/186/4/final

Introduced in R2018a

Nonsecure RSA public exponent

Context used in key generation is associated with low exponent value

Description

This defect occurs when you attempt RSA key generation by using a context object that is associated with a low public exponent.

For instance, you set a public exponent of 3 in the context object, and then use it for key generation.

```
/* Set public exponent */
ret = BN_dec2bn(&pubexp, "3");

/* Initialize context */
ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
pkey = EVP_PKEY_new();
ret = EVP_PKEY_keygen_init(kctx);

/* Set public exponent in context */
ret = EVP_PKEY_CTX_set_rsa_keygen_pubexp(ctx, pubexp);

/* Generate key */
ret = EVP_PKEY_keygen(kctx, &pkey);
```

Risk

A low RSA public exponent makes certain kinds of attacks more damaging, especially when a weak padding scheme is used or padding is not used at all.

Fix

It is recommended to use a public exponent of 65537. Using a higher public exponent can make the operations slower.

Examples

Using RSA Public Exponent of 3

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY *pkey){
  BIGNUM* pubexp;
  EVP_PKEY_CTX* ctx;

  pubexp = BN_new();
  if (pubexp == NULL) fatal_error();
  ret = BN_set_word(pubexp, 3);
```

```
  if (ret <= 0) fatal_error();

  ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 2048);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_pubexp(ctx, pubexp);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_keygen(ctx, &pkey);
}
```

In this example, an RSA public exponent of 3 is associated with the context object `ctx`. The low exponent makes operations that use the generated key vulnerable to certain attacks.

Correction — Use Public Exponent of 65537

One possible correction is to use the recommended public exponent 65537.

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY *pkey){
  BIGNUM* pubexp;
  EVP_PKEY_CTX* ctx;

  pubexp = BN_new();
  if (pubexp == NULL) fatal_error();
  ret = BN_set_word(pubexp, 65537);
  if (ret <= 0) fatal_error();

  ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 2048);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_pubexp(ctx, pubexp);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_keygen(ctx, &pkey);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_RSA_LOW_EXPONENT`
Impact: Medium

CWE ID: 310, 326, 327, 522

See Also

`Find defects (-checkers)` | `Incompatible padding for RSA algorithm operation` | `Missing blinding for RSA algorithm` | `Missing padding for RSA algorithm` | `Weak padding for RSA algorithm`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Nonsecure SSL/TLS protocol

Context used for handling SSL/TLS connections is associated with weak protocol

Description

This defect occurs when you do not disable nonsecure protocols in an `SSL_CTX` or `SSL` context object before using the object for handling SSL/TLS connections.

For instance, you disable the protocols SSL2.0 and TLS1.0 but forget to disable the protocol SSL3.0, which is also considered weak.

```
/* Create and configure context */
ctx = SSL_CTX_new(SSLv23_method());
SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2|SSL_OP_NO_TLSv1);

/* Use context to handle connection */
ssl = SSL_new(ctx);
SSL_set_fd(ssl, NULL);
ret = SSL_connect(ssl);
```

Risk

The protocols SSL2.0, SSL3.0, and TLS1.0 are considered weak in the cryptographic community. Using one of these protocols can expose your connections to cross-protocol attacks. The attacker can decrypt an RSA ciphertext without knowing the RSA private key.

Fix

Disable the nonsecure protocols in the context object before using the object to handle connections.

```
/* Create and configure context */
ctx = SSL_CTX_new(SSLv23_method());
SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2|SSL_OP_NO_SSLv3|SSL_OP_NO_TLSv1);
```

Examples

Nonsecure Protocols Not Disabled

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <openssl/ssl.h>
#include <openssl/err.h>


#define fatal_error() exit(-1)

int ret;
int func(){
  SSL_CTX *ctx;
```

```
  SSL *ssl;

  SSL_library_init();

  /* context configuration */
  ctx = SSL_CTX_new(SSLv23_client_method());
  if (ctx==NULL) fatal_error();

  ret = SSL_CTX_use_certificate_file(ctx, "cert.pem", SSL_FILETYPE_PEM);
  if (ret <= 0) fatal_error();

  ret = SSL_CTX_load_verify_locations(ctx, NULL, "ca/path");
  if (ret <= 0) fatal_error();

  /* Handle connection */
  ssl = SSL_new(ctx);
  if (ssl==NULL) fatal_error();
  SSL_set_fd(ssl, NULL);

  return SSL_connect(ssl);
}
```

In this example, the protocols SSL2.0, SSL3.0, and TLS1.0 are not disabled in the context object before the object is used for a new connection.

Correction — Disable Nonsecure Protocols

Disable nonsecure protocols before using the objects for a new connection. Use the function `SSL_CTX_set_options` to disable the protocols SSL2.0, SSL3.0, and TLS1.0.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <openssl/ssl.h>
#include <openssl/err.h>


#define fatal_error() exit(-1)

int ret;
int func(){
  SSL_CTX *ctx;
  SSL *ssl;

  SSL_library_init();

  /* context configuration */
  ctx = SSL_CTX_new(SSLv23_client_method());
  if (ctx==NULL) fatal_error();

  SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2|SSL_OP_NO_SSLv3|SSL_OP_NO_TLSv1);

  ret = SSL_CTX_use_certificate_file(ctx, "cert.pem", SSL_FILETYPE_PEM);
  if (ret <= 0) fatal_error();
```

```
  ret = SSL_CTX_load_verify_locations(ctx, NULL, "ca/path");
  if (ret <= 0) fatal_error();

  /* Handle connection */
  ssl = SSL_new(ctx);
  if (ssl==NULL) fatal_error();
  SSL_set_fd(ssl, NULL);

  return SSL_connect(ssl);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_SSL_WEAK_PROTOCOL`
Impact: Medium
CWE ID: 310, 327, 522, 693

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

Predictable block cipher initialization vector

Initialization vector is generated from a weak random number generator

Description

This defect occurs when you use a weak random number generator for the block cipher initialization vector.

Risk

If you use a weak random number generator for the initiation vector, your data is vulnerable to dictionary attacks.

Block ciphers break your data into blocks of fixed size. Block cipher modes such as CBC (Cipher Block Chaining) protect against dictionary attacks by XOR-ing each block with the encrypted output from the previous block. To protect the first block, these modes use a random initialization vector (IV). If you use a weak random number generator for your IV, your data becomes vulnerable to dictionary attacks.

Fix

Use a strong pseudo-random number generator (PRNG) for the initialization vector. For instance, use:

- OS-level PRNG such as `/dev/random` on UNIX or `CryptGenRandom()` on Windows
- Application-level PRNG such as Advanced Encryption Standard (AES) in Counter (CTR) mode, HMAC-SHA1, etc.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

Examples

Predictable Initialization Vector

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16];
    RAND_pseudo_bytes(iv, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

In this example, the function `RAND_pseudo_bytes` declared in `openssl/rand.h` produces the initialization vector. The byte sequences that `RAND_pseudo_bytes` generates are not necessarily unpredictable.

Correction — Use Strong Random Number Generator

Use a strong random number generator to produce the initialization vector. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16];
    RAND_bytes(iv, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_CIPHER_PREDICTABLE_IV`
Impact: Medium
CWE ID: 310, 329, 330, 338

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Predictable cipher key

Encryption or decryption key is generated from a weak random number generator

Description

This defect occurs when you use a weak random number generator for the encryption or decryption key.

Risk

If you use a weak random number generator for the encryption or decryption key, an attacker can retrieve your key easily.

You use a key to encrypt and later decrypt your data. If a key is easily retrieved, data encrypted using that key is not secure.

Fix

Use a strong pseudo-random number generator (PRNG) for the key. For instance:

- Use an OS-level PRNG such as `/dev/random` on UNIX or `CryptGenRandom()` on Windows
- Use an application-level PRNG such as Advanced Encryption Standard (AES) in Counter (CTR) mode, HMAC-SHA1, etc.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

Examples

Predictable Cipher Key

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16];
    RAND_pseudo_bytes(key, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

In this example, the function `RAND_pseudo_bytes` declared in `openssl/rand.h` produces the cipher key. However, the byte sequences that `RAND_pseudo_bytes` generates are not necessarily unpredictable.

Correction — Use Strong Random Number Generator

One possible correction is to use a strong random number generator to produce the cipher key. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16];
    RAND_bytes(key, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_CIPHER_PREDICTABLE_KEY`
Impact: Medium
CWE ID: 310, 326, 330, 338

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Server certificate common name not checked

Attacker might use valid certificate to impersonate trusted host

Description

The defect occurs when you do not check the common name provided in the server certificate against the domain name of the server.

Typically, when a client connects to a server, the server sends a digital certificate to the client that identifies the server as a trusted entity. The certificate contains information about the server, including the common name of the server. The common name matches the server domain name that the certificate identifies as a trusted entity.

The checker raises no defect if:

- You pass the SSL context as an argument to the function that calls `SSL_new`.
- You declare the SSL context outside the scope of the function handling the connection.

Risk

A malicious attacker might use a valid certificate to impersonate a trusted host, resulting in the client interacting with an untrusted server.

Fix

Use one of these functions to specify the server domain name that the program checks against the common name provided in the server certificate.

- `SSL_set_tlsext_host_name`
- `SSL_set1_host`
- `SSL_add1_host`

Examples

Client Checks Server Certificate but not Server Domain Name

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

#define fatal_error() exit(-1)

void check_certificate(SSL_CTX* ctx, SSL* ssl)
{
    /* Check for Client authentication error */
    if (!SSL_get_peer_certificate(ssl)) {
        printf("SSL Client Authentication error\n");
        SSL_free(ssl);
        SSL_CTX_free(ctx);
```

```
        exit(0);
    }
    /* Check for Client authentication error */
    if (SSL_get_verify_result(ssl) != X509_V_OK) {
        printf("SSL Client Authentication error\n");
        SSL_free(ssl);
        SSL_CTX_free(ctx);
        exit(0);
    }
}

void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;

    /* creation context for the SSL protocol */
    ctx = SSL_CTX_new(SSLv23_client_method());
    if (ctx == NULL) fatal_error();

    /* Handle connection */
    ssl = SSL_new(ctx);
    SSL_set_connect_state(ssl);
    check_certificate(ctx, ssl);
    ret = SSL_connect(ssl);
    if (ret <= 0) fatal_error();

    SSL_free(ssl);
    SSL_CTX_free(ctx);
}
```

In this example, an SSL structure is initiated with a client connection method. The client validates the server certificate with `check_certificate`. However, the client does not check that the certificate common name matches the domain name of the server. An attacker might use the valid certificate to impersonate the trusted server.

Correction — Specify a Domain Name to Check Against the Certificate Common Name

One possible correction is to use `SSL_set1_host` to specify the expected domain name that the program checks against the server certificate common name.

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

#define fatal_error() exit(-1)

void check_certificate(SSL_CTX* ctx, SSL* ssl)
{
    /* Check for Client authentication error */
    if (!SSL_get_peer_certificate(ssl)) {
        printf("SSL Client Authentication error\n");
        SSL_free(ssl);
        SSL_CTX_free(ctx);
        exit(0);
```

```
    }
    /* Check for Client authentication error */
    if (SSL_get_verify_result(ssl) != X509_V_OK) {
        printf("SSL Client Authentication error\n");
        SSL_free(ssl);
        SSL_CTX_free(ctx);
        exit(0);
    }
}

void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;

    /* creation context for the SSL protocol */
    ctx = SSL_CTX_new(SSLv23_client_method());
    if (ctx == NULL) fatal_error();

    /* Handle connection */
    ssl = SSL_new(ctx);
    SSL_set_connect_state(ssl);
    check_certificate(ctx, ssl);
    ret = SSL_set1_host(ssl, "www.mysite.com");
    if (ret <= 0) fatal_error();
    ret = SSL_connect(ssl);
    if (ret <= 0) fatal_error();

    SSL_free(ssl);
    SSL_CTX_free(ctx);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_SSL_HOSTNAME_NOT_CHECKED`
Impact: Medium
CWE ID: 297

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

TLS/SSL connection method not set

Program cannot determine whether to call client or server routines

Description

The defect occurs when you call one of these functions without explicitly setting the connection method of the TLS/SSL context.

- `SSL_read`
- `SSL_write`
- `SSL_do_handshake`

The communication between server and client entities that use a TLS/SSL connection begins with a handshake. During the handshake, the parties exchange information and establish the encryption algorithm and session keys the parties use during the session. The connection methods for the server and client use different routines for the handshake.

The checker raises no defect if:

- You use `SSL_connect` (client) and `SSL_accept` (server) functions. These functions set the correct handshake routines automatically.
- You pass the SSL context as an argument to the function that calls `SSL_new`.
- You declare the SSL context outside the scope of the function handling the connection.

Risk

You cannot begin a handshake if the SSL engine does not know which connection method routines to call.

Fix

- For client handshake routines, call `SSL_set_connect_state` before you begin the handshake.
- For server handshake routines, call `SSL_set_accept_state` before you begin the handshake.

Examples

Server Connection Method Not Set Explicitly

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

#define fatal_error() exit(-1)

int len;
unsigned char buf;
volatile int rd;

const SSL_METHOD*  set_method()
```

```
{
    return SSLv23_server_method();
}

void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;
    const SSL_METHOD* method =  set_method();
    ctx = SSL_CTX_new(method);
    ssl = SSL_new(ctx);

    switch (rd) {
    case 1:
        ret = SSL_read(ssl, (void*)buf, len);
        if (ret <= 0) fatal_error();
        break;
    case 2:
        ret = SSL_do_handshake(ssl);
        if (ret <= 0) fatal_error();
        break;
    default:
        ret = SSL_write(ssl, (void*)buf, len);
        if (ret <= 0) fatal_error();
        break;
    }
}
```

In this example, the SSL context `ctx` is generated with server connection method `SSLv23_server_method`. However, the connection method is not set explicitly for the SSL structure `ssl` before the attempt to read from the connection, initiate a handshake, or write to the connection.

Correction — Set Server Connection Method Explicitly

One possible correction is to call `SSL_set_accept_state` to set the server role for the SSL structure `ssl` before you begin the handshake.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

#define fatal_error() exit(-1)

int len;
unsigned char buf;
volatile int rd;

const SSL_METHOD*  set_method()
{
    return SSLv23_server_method();
}

void func()
{
    int ret;
    SSL_CTX* ctx;
```

```
    SSL* ssl;
    const SSL_METHOD* method =  set_method();
    ctx = SSL_CTX_new(method);
    ssl = SSL_new(ctx);
    SSL_set_accept_state(ssl);


    switch (rd) {
    case 1:
        ret = SSL_read(ssl, (void*)buf, len);
        if (ret <= 0) fatal_error();
        break;
    case 2:
        ret = SSL_do_handshake(ssl);
        if (ret <= 0) fatal_error();
        break;
    default:
        ret = SSL_write(ssl, (void*)buf, len);
        if (ret <= 0) fatal_error();
        break;
    }
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_SSL_NO_ROLE`
Impact: Medium
CWE ID: 304, 322, 573

See Also

`Find defects (-checkers)|Missing X.509 certificate|Missing certification authority list`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

TLS/SSL connection method set incorrectly

Program calls functions that do not match role set by connection method

Description

The defect occurs when you call functions that do not match the role set by the connection method that you specified for the SSL context.

The functions that you call when handling a TLS/SSL connection between client and server entities are different, depending on the role of the entity. For instance, the connection between a server and a client begins with a handshake. The client always initiates the handshake. You use `SSL_accept` with a server entity to wait for a client to initiate the handshake.

Typically, you set a connection method when you initiate the SSL context. The method specifies the role of the entity.

The checker flags the use of functions that do not match the connection method specified for the SSL context. See the **Event** column in the **Results Details** pane to view connection method specified for the SSL context.

Risk

If you set the TLS/SSL connection method incorrectly, the functions you use to handle the connection might not match the role specified by the method. For instance, you use `SSL_accept` with a client entity to wait for a client to initiate a handshake instead of `SSL_connect` to initiate the handshake with a server.

Fix

Make sure that you use functions that match the TLS/SSL connection method to handle the connection.

Examples

Client Waiting for Client to Initiate Handshake

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

#define fatal_error() exit(-1)

const SSL_METHOD*  set_method()
{
    return SSLv23_client_method();
}

void set_method_1(SSL* ssl)
{
    SSL_set_connect_state(ssl);
}
```

```
void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;
    const SSL_METHOD* method = set_method();
    ctx = SSL_CTX_new(method);
    ssl = SSL_new(ctx);
    set_method_1(ssl);
    ret = SSL_accept(ssl);
    if (ret <= 0) fatal_error();
}
```

In this example, the SSL context `ctx` is initialized with a client role. The SSL structure is also explicitly set to client role through the call to `set_method_1`. To establish a connection with the server, the client should initiate a handshake with the server. Instead, `SSL_accept` causes the client to wait for another client to initiate a handshake.

Correction — Use `SSL_connect` to Initiate Handshake with Server

One possible correction is to use `SSL_connect` to initiate the TLS/SSL handshake with the server.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

#define fatal_error() exit(-1)

const SSL_METHOD*  set_method()
{
    return SSLv23_client_method();
}

void set_method_1(SSL* ssl)
{
    SSL_set_connect_state(ssl);
}
void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;
    const SSL_METHOD* method = set_method();
    ctx = SSL_CTX_new(method);
    ssl = SSL_new(ctx);
    set_method_1(ssl);
    ret = SSL_connect(ssl);
    if (ret <= 0) fatal_error();
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_SSL_BAD_ROLE`

Impact: Medium
CWE ID: 573

See Also

`Find defects (-checkers)|Missing X.509 certificate|Missing certification authority list|TLS/SSL connection method not set`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Weak cipher algorithm

Encryption algorithm associated with the cipher context is weak

Description

This defect occurs when you associate a weak encryption algorithm with the cipher context.

Risk

Some encryption algorithms have known flaws. Though the OpenSSL library still supports the algorithms, you must avoid using them.

If your cipher algorithm is weak, an attacker can decrypt your data by exploiting a known flaw or brute force attacks.

Fix

Use algorithms that are well-studied and widely acknowledged as secure.

For instance, the Advanced Encryption Standard (AES) is a widely accepted cipher algorithm.

Examples

Use of DES Algorithm

```
#include <openssl/evp.h>
#include <stdlib.h>

void func(unsigned char *key, unsigned char *iv) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);
    const EVP_CIPHER * ciph = EVP_des_cbc();
    EVP_EncryptInit_ex(ctx, ciph, NULL, key, iv);
}
```

In this example, the routine `EVP_des_cbc()` invokes the Data Encryption Standard (DES) algorithm, which is now considered as non-secure and relatively slow.

Correction — Use AES Algorithm

One possible correction is to use the faster and more secure Advanced Encryption Standard (AES) algorithm instead.

```
#include <openssl/evp.h>
#include <stdlib.h>

void func(unsigned char *key, unsigned char *iv) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);
```

```
    const EVP_CIPHER * ciph = EVP_aes_128_cbc();
    EVP_EncryptInit_ex(ctx, ciph, NULL, key, iv);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_CIPHER_WEAK_CIPHER`
Impact: Medium
CWE ID: 310, 326, 327

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Weak cipher mode

Encryption mode associated with the cipher context is weak

Description

This defect occurs when you associate a weak block cipher mode with the cipher context.

The cipher mode that is especially flagged by this defect is the Electronic Code Book (ECB) mode.

Risk

The ECB mode does not support protection against dictionary attacks.

An attacker can decrypt your data even using brute force attacks.

Fix

Use a cipher mode more secure than ECB.

For instance, the Cipher Block Chaining (CBC) mode protects against dictionary attacks by:

- XOR-ing each block of data with the encrypted output from the previous block.
- XOR-ing the first block of data with a random initialization vector (IV).

Examples

Use of ECB Mode

```
#include <openssl/evp.h>
#include <stdlib.h>

void func(unsigned char *key, unsigned char *iv) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);
    const EVP_CIPHER * ciph = EVP_aes_128_ecb();
    EVP_EncryptInit_ex(ctx, ciph, NULL, key, iv);
}
```

In this example, the routine `EVP_aes_128_ecb()` invokes the Advanced Encryption Standard (AES) algorithm in the Electronic Code Book (ECB) mode. The ECB mode does not support protection against dictionary attacks.

Correction — Use CBC Mode

One possible correction is to use the Cipher Block Chaining (CBC) mode instead.

```
#include <openssl/evp.h>
#include <stdlib.h>
```

```
void func(unsigned char *key, unsigned char *iv) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);
    const EVP_CIPHER * ciph = EVP_aes_128_cbc();
    EVP_EncryptInit_ex(ctx, ciph, NULL, key, iv);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_CIPHER_WEAK_MODE`
Impact: Medium
CWE ID: 310, 326, 327

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2017a

Weak padding for RSA algorithm

Context used in encryption or signing operation is associated with insecure padding type

Description

This defect occurs when you perform RSA encryption or signature by using a context object that was previously associated with a weak padding scheme.

For instance, you perform encryption by using a context object that is associated with the PKCS#1v1.5 padding scheme. The scheme is considered insecure and has already been broken.

```
ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_PADDING);
...
ret = EVP_PKEY_encrypt(ctx, out, &out_len, in, in_len)
```

Risk

Padding schemes remove determinism from the RSA algorithm and protect RSA operations from certain kinds of attacks. Padding schemes such as PKCS#1v1.5, ANSI X9.31, and SSLv23 are known to be vulnerable. Do not use these padding schemes for encryption or signature operations.

Fix

Before performing an RSA operation, associate the context object with a strong padding scheme.

- Encryption: Use the OAEP padding scheme.

  For instance, use the `EVP_PKEY_CTX_set_rsa_padding` function with the argument `RSA_PKCS1_OAEP_PADDING` or the `RSA_padding_add_PKCS1_OAEP` function.

  ```
  ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_PADDING);
  ```

  You can then use functions such as `EVP_PKEY_encrypt` / `EVP_PKEY_decrypt` or `RSA_public_encrypt` / `RSA_private_decrypt` on the context.
- Signature: Use the RSA-PSS padding scheme.

  For instance, use the `EVP_PKEY_CTX_set_rsa_padding` function with the argument `RSA_PKCS1_PSS_PADDING`.

  ```
  ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_PSS_PADDING);
  ```

  You can then use functions such as the `EVP_PKEY_sign`-`EVP_PKEY_verify` pair or the `RSA_private_encrypt`-`RSA_public_decrypt` pair on the context.

Examples

Encryption with PKCS#1v1.5 Padding

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>
```

```
#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;

int func(unsigned char *src, size_t len, RSA* rsa){
  if (rsa == NULL) fatal_error();

  return RSA_public_encrypt(len, src, out_buf, rsa, RSA_PKCS1_PADDING);
}
```

In this example, the PKCS#1v1.5 padding scheme is used in the encryption step.

Correction — Use OAEP Padding Scheme

Use the OAEP padding scheme for stronger encryption.

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;

int func(unsigned char *src, size_t len, RSA* rsa){
  if (rsa == NULL) fatal_error();

  return RSA_public_encrypt(len, src, out_buf, rsa, RSA_PKCS1_OAEP_PADDING);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: `CRYPTO_RSA_WEAK_PADDING`
Impact: Medium
CWE ID: 310, 326, 327, 780

See Also

`Find defects (-checkers)` | `Incompatible padding for RSA algorithm operation` | `Missing blinding for RSA algorithm` | `Missing padding for RSA algorithm` | `Nonsecure RSA public exponent`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018a

X.509 peer certificate not checked

Connection might be vulnerable to man-in-the-middle attacks

Description

The defect occurs when you do not properly check the X.509 certificate used to authenticate the TLS/SSL connection when handling the connection. To properly check the certificate, you must call these two functions together to obtain and verify the certificate.

- `SSL_get_peer_certificate`: Obtains a certificate from the client or server you are trying to authenticate. The function returns NULL if no certificate is present. Even if the function returns a certificate, the certificate must still be checked.
- `SSL_get_verify_result`: Verifies the certificate presented by the client or server. If you do not obtain a certificate before calling this function, there are no verification errors and the function returns successfully.

The checker raises a defect on the functions `SSL_read` or `SSL_write` when you attempt to read from or write to the TLS/SSL connection.

The checker raises no defect if:

- You declare the SSL context outside the scope of the function handling the connection.
- You use anonymous cypher suites.

Risk

If you do not properly check the validity of the certificate of the peer you are attempting to authenticate, your connection is vulnerable to man-in-the-middle attacks.

Fix

To properly check the validity of the certificate, call both `SSL_get_peer_certificate` and `SSL_get_verify_result`.

Examples

Client Certificate Obtained But Not Verified

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

#define fatal_error() exit(-1)

int len;
unsigned char buf;

void func()
```

```
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;

    /* creation context for the SSL protocol */
    ctx = SSL_CTX_new(SSLv23_client_method());
    if (ctx == NULL) fatal_error();

    /* Set to require peer (client) certificate */
    SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);

    /* Handle connection */
    ssl = SSL_new(ctx);
    if (ssl == NULL) fatal_error();
    ret = SSL_set_fd(ssl, NULL);
    if (!ret) fatal_error();
    ret = SSL_connect(ssl);
    if (ret <= 0) fatal_error();

    /* Check for Client authentication error */
    if (!SSL_get_peer_certificate(ssl)) {
        printf("SSL Client Authentication error\n");
        SSL_free(ssl);
        SSL_CTX_free(ctx);
        exit(0);
    }

    /*Read message from the client.*/
    ret = SSL_read(ssl, (void*)buf, len);
    if (ret <= 0) fatal_error();

    /* Close connection */
    SSL_free(ssl);
    SSL_CTX_free(ctx);
}
```

In this example, a TLS/SSL context is created for a server connection method. The function `SSL_get_peer_certificate` then requests the client certificate to authenticate the connection. However, the server then attempts to read from the connection without checking the validity of the returned certificate. The certificate might be invalid, and the connection could be vulnerable to a man-in-the-middle attack.

Correction — Check the Validity of the Returned Certificate

One possible correction is to check the validity of the returned certificate by calling `SSL_get_verify_result`.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

#define fatal_error() exit(-1)

int len;
```

```
unsigned char buf;

void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;

    /* creation context for the SSL protocol */
    ctx = SSL_CTX_new(SSLv23_client_method());
    if (ctx == NULL) fatal_error();

    /* Set to require peer (client) certificate */
    SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);

    /* Handle connection */
    ssl = SSL_new(ctx);
    if (ssl == NULL) fatal_error();
    ret = SSL_set_fd(ssl, NULL);
    if (!ret) fatal_error();
    ret = SSL_connect(ssl);
    if (ret <= 0) fatal_error();

    /* Check for Client authentication error */
    if (!SSL_get_peer_certificate(ssl)) {
        printf("SSL Client Authentication error\n");
        SSL_free(ssl);
        SSL_CTX_free(ctx);
        exit(0);
    }

    if (SSL_get_verify_result(ssl) != X509_V_OK) {
        printf("SSL Client Authentication error\n");
        SSL_free(ssl);
        SSL_CTX_free(ctx);
        exit(0);
    }

    /*Read message from the client.*/
    ret = SSL_read(ssl, (void*)buf, len);
    if (ret <= 0) fatal_error();

    /* Close connection */
    SSL_free(ssl);
    SSL_CTX_free(ctx);
}
```

Result Information

Group: Cryptography
Language: C | C++
Default: Off
Command-Line Syntax: CRYPTO_SSL_CERT_NOT_CHECKED
Impact: Medium
CWE ID: 287

See Also

`Find defects (-checkers)|Missing X.509 certificate|Missing certification authority list`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Tainted Data Defects

Array access with tainted index

Array index from unsecure source possibly outside array bounds

Description

This defect occurs when you access an array by using a tainted index that has not been validated.

Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

Examples

Use Index to Return Buffer Value

```
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    return tab[num];
}
```

In this example, the index `num` accesses the array `tab`. The function does not check to see if `num` is inside the range of `tab`.

Correction — Check Range Before Use

One possible correction is to check that `num` is in range before using it.

```
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    if (num >= 0 && num < SIZE100) {
            return tab[num];
    } else {
        return -9999;
    }
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: `TAINTED_ARRAY_INDEX`
Impact: Medium
CWE ID: 121, 124, 125, 129

See Also

`Loop bounded with tainted value` | `Pointer dereference with tainted offset` | `Tainted size of variable length array` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Command executed from externally controlled path

Path argument from an unsecure source

Description

This defect occurs when the path to a command executed in the program is constructed from external sources.

Risk

An attacker can:

- Change the command that the program executes, possibly to a command that only the attack can control.
- Change the environment in which the command executes, by which the attacker controls what the command means and does.

Fix

Before calling the command, validate the path to make sure that it is the intended location.

Examples

Executing Path from Environment Variable

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedpathcmd() {
    char cmd[SIZE128] = "";
    char* userpath = getenv("MYAPP_PATH");

    strncpy(cmd, userpath, SIZE100);
    strcat(cmd, "/ls *");
    /* Launching command */
    system(cmd);
}
```

This example obtains a path from an environment variable `MYAPP_PATH`. `system` runs a command from that path without checking the value of the path. If the path is not the intended path, your program executes in the wrong location.

Correction — Use Trusted Path

One possible correction is to use a list of allowed paths to match against the environment variable path.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

/* Function to sanitize a string */
int sanitize_str(char* s, size_t n) {
    int res = 0;
    /* String is ok if */
    if (s && n>0 && n<SIZE128) {
        /* - string is not null                       */
        /* - string has a positive and limited size */
        s[n-1] = '\0';  /* Add a security \0 char at end of string */
        /* Tainted pointer detected above, used as "firewall" */
        res = 1;
    }
    return res;
}

/* Authorized path ids */
enum { PATH0=1, PATH1, PATH2 };

void taintedpathcmd() {
    char cmd[SIZE128] = "";

    char* userpathid = getenv("MYAPP_PATH_ID");
    if (sanitize_str(userpathid, SIZE100)) {
        int pathid = atoi(userpathid);


        char path[SIZE128] = "";
        switch(pathid) {
            case PATH0:
                strcpy(path, "/usr/local/my_app0");
                break;
            case PATH1:
                strcpy(path, "/usr/local/my_app1");
                break;
            case PATH2:
                strcpy(path, "/usr/local/my_app2");
                break;
            default:
                /* do nothing */
        break;
        }
        if (strlen(path)>0) {
            strncpy(cmd, path, SIZE100);
            strcat(cmd, "/ls *");
```

```
            system(cmd);
        }
    }
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: TAINTED_PATH_CMD
Impact: Medium
CWE ID: 114, 426

See Also

`Execution of externally controlled command` | `Use of externally controlled environment variable` | `Host change using externally controlled elements` | `Library loaded from externally controlled path` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Execution of externally controlled command

Command argument from an unsecure source vulnerable to operating system command injection

Description

This defect occurs when commands are fully or partially constructed from externally controlled input.

Risk

Attackers can use the externally controlled input as operating system commands, or arguments to the application. An attacker could read or modify sensitive data can be read or modified, execute unintended code, or gain access to other aspects of the program.

Fix

Validate the inputs to allow only intended input values. For example, create a whitelist of acceptable inputs and compare the input against this list.

Examples

Call Argument Command

```
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "unistd.h"
#include "dlfcn.h"
#include "limits.h"

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void taintedexternalcmd(char* usercmd)
{
    char cmd[SIZE128] = "/usr/bin/cat ";
    strcat(cmd, usercmd);
    system(cmd);
}
```

This example function calls a command from a user argument without checking the command variable.

Correction — Use a Predefined Command

One possible correction is to use a `switch` statement to run a predefined command, using the user input as the switch variable.

```
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "unistd.h"
#include "dlfcn.h"
#include "limits.h"

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
enum { CMD0 = 1, CMD1, CMD2 };

void taintedexternalcmd(int usercmd)
{
    char cmd[SIZE128] = "/usr/bin/cat ";

    switch(usercmd) {
        case CMD0:
            strcat(cmd, "*.c");
            break;
        case CMD1:
            strcat(cmd, "*.h");
            break;
        case CMD2:
            strcat(cmd, "*.cpp");
            break;
        default:
            strcat(cmd, "*.c");
    }
    system(cmd);
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: TAINTED_EXTERNAL_CMD
Impact: Medium
CWE ID: 77, 78, 88, 114

See Also

`Use of externally controlled environment variable` | `Host change using externally controlled elements` | `Command executed from externally controlled path` | `Library loaded from externally controlled path` | `Execution of a binary from a relative path can be controlled by an external actor` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Host change using externally controlled elements

Changing host ID from an unsecure source

Description

This defect occurs when routines that change the host ID, such as `sethostid` (Linux) or `SetComputerName` (Windows), use arguments that are externally controlled.

Risk

The tainted host ID value can allow external control of system settings. This control can disrupt services, cause unexpected application behavior, or cause other malicious intrusions.

Fix

Use caution when changing or editing the host ID. Do not allow user-provided values to control sensitive data.

Examples

Change Host ID from Function Argument

```
#include <unistd.h>

void bug_taintedhostid(long userhid) {
    sethostid(userhid);
}
```

This example sets a new host ID using the argument passed to the function. Before using the host ID, check the value passed in.

Correction — Predefined Host ID

One possible correction is to change the host ID to a predefined ID. This example uses the host argument as a switch variable to choose between the different, predefined host IDs.

```
#include <unistd.h>

extern long called_taintedhostid_sanitize(long);
enum { HI0 = 1, HI1, HI2, HI3 };

void taintedhostid(int host) {

    long hid = 0;
    switch(host) {
        case HI0:
            hid = 0x7f0100;
            break;
        case HI1:
            hid = 0x7f0101;
            break;
        case HI2:
            hid = 0x7f0102;
```

```
            break;
        case HI3:
            hid = 0x7f0103;
            break;
        default:
            /* do nothing */
        break;
    }
    if (hid > 0) {
        sethostid(hid);
    }
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: `TAINTED_HOSTID`
Impact: Medium
CWE ID: 15

See Also

`Execution of externally controlled command|Use of externally controlled environment variable|Host change using externally controlled elements|Command executed from externally controlled path|Library loaded from externally controlled path|Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Library loaded from externally controlled path

Using a library argument from an externally controlled path

Description

This defect occurs when libraries are loaded from fixed or externally controlled paths and unintended actors can control one or more locations on the paths.

Risk

If an attacker knows or controls the path that you use to load a library, the attacker can change:

- The library that the program loads, replacing the intended library and commands.
- The environment in which the library executes, giving unintended permissions and capabilities to the attacker.

Fix

When possible, use hard-coded or fully qualified path names to load libraries. It is possible the hard-coded paths do not work on other systems. Use a centralized location for hard-coded paths, so that you can easily modify the path within the source code.

Another solution is to use functions that require explicit paths. For example, `system()` does not require a full path because it can use the `PATH` environment variable. However, `execl()` and `execv()` do require the full path.

Examples

Call Custom Library

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dlfcn.h>
#include <limits.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void* taintedpathlib() {
    void* libhandle = NULL;
    char lib[SIZE128] = "";
    char* userpath = getenv("LD_LIBRARY_PATH");
    strncpy(lib, userpath, SIZE128);
    strcat(lib, "/libX.so");
    libhandle = dlopen(lib, 0x00001);
    return libhandle;
}
```

This example loads the library `libX.so` from an environment variable `LD_LIBRARY_PATH`. An attacker can change the library path in this environment variable. The actual library you load could be a different library from the one that you intend.

Correction — Change and Check Path

One possible correction is to change how you get the library path and check the path of the library before opening the library. This example receives the path as an input argument but then performs the following checks on the path:

- The function `sanitize_str` protects against possible buffer overflows.
- The function `identified_safe_libX_folder` checks if the path belongs to a list of whitelisted paths.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dlfcn.h>
#include <limits.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

/* Use white list */
static const char *libX_safe_folder[] = {
  "/usr/",
  "/usr/lib",
  "/lib"
};

/* Return the index if the input is in the white list */
int identified_safe_libX_folder(const char* path)
{
  for (int i = 0; i < sizeof(libX_safe_folder) / sizeof(libX_safe_folder[0]); i ++)
    {
      if (strcmp(path, libX_safe_folder[i]) == 0)
      return i;
    }
  return -1;
}

/* Function to sanitize a string */
char *sanitize_str(char* s, size_t n) {
  /* strlen is used here as a kind of firewall for tainted string errors */
  if (strlen(s) > 0 && strlen(s) < n)
    return s;
  else
    return NULL;
}

void* taintedpathlib(char* userpath) {
  void* libhandle = NULL;
```

```
  const char *const checked_userpath = sanitize_str(userpath, SIZE128);
  if (checked_userpath != NULL) {
    int index = identified_safe_libX_folder(checked_userpath);
    if (index > 0) {
      char lib[SIZE128] = "";
      strncpy(lib, libX_safe_folder[index], SIZE128);
      strcat(lib, "/libX.so");
      libhandle = dlopen(lib, RTLD_LAZY);
    }
  }
  return libhandle;
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: `TAINTED_PATH_LIB`
Impact: Medium
CWE ID: 114, 426

See Also

`Execution of externally controlled command` | `Use of externally controlled environment variable` | `Command executed from externally controlled path` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Loop bounded with tainted value

Loop controlled by a value from an unsecure source

Description

This defect occurs when a loop is bounded by values from nonsecure sources.

Risk

A tainted value can cause over looping or infinite loops. Attackers can use this vulnerability to crash your program or cause other unintended behavior.

Fix

Before starting the loop, validate unknown boundary and iterator values.

Examples

Loop Boundary From Input Argument

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedloopboundary(int count) {
    int res = 0;
    for (int i=0 ; i < count; ++i) {
        res += i;
    }
    return res;
}
```

In this example, the function uses the input argument to loop `count` times. `count` could be any number because the value is not checked before starting the for-loop.

Correction — Check Loop Control

One possible correction is to check the value of the variable controlling the loop before starting the for-loop. This example checks if `count` is greater than zero and less than the maximum size.

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedloopboundary(int count) {
    int res = 0;

    if (count>0 && count<SIZE128) {
        for (int i=0 ; i<count ; ++i) {
```

```
            res += i;
        }
    }
    return res;
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: TAINTED_LOOP_BOUNDARY
Impact: Medium
CWE ID: 606

See Also

Array access with tainted index | Pointer dereference with tainted offset | Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Memory allocation with tainted size

Size argument to memory function is from an unsecure source

Description

This defect occurs when a memory allocation function, such as `calloc` or `malloc`, uses a size argument from a nonsecure source.

Risk

Uncontrolled memory allocation can cause your program to request too much system memory. This consequence can lead to a crash due to an out-of-memory condition, or assigning too many resources.

Fix

Before allocating memory, check the value of your arguments to check that they do not exceed the bounds.

Examples

Allocate Memory Using Input Argument

```
#include "stdlib.h"

int* bug_taintedmemoryallocsize(size_t size) {
    int* p = (int*)malloc(size);
    return p;
}
```

In this example, `malloc` allocates `size` amount of memory for the pointer `p`. `size` is an outside variable, so could be any size value. If the size is larger than the amount of memory you have available, your program could crash.

Correction — Check Size of Memory to be Allocated

One possible correction is to check the size of the memory that you want to allocate before performing the `malloc` operation. This example checks to see if the size is positive and less than the maximum size.

```
#include "stdlib.h"

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int* corrected_taintedmemoryallocsize(int size) {
    int* p = NULL;
    if (size>0 && size<SIZE128) {          /* Fix: Check entry range before use */
        p = (int*)malloc((unsigned int)size);
```

```
    }
    return p;
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: `TAINTED_MEMORY_ALLOC_SIZE`
Impact: Medium
CWE ID: 128, 131, 789

See Also

`Unprotected dynamic memory allocation` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Pointer dereference with tainted offset

Offset is from an unsecure source and dereference may be out of bounds

Description

This defect occurs when a pointer dereference uses an offset variable from an unknown or nonsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

Fix

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

Examples

Dereference Pointer Array

```
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(int i) {
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[i];
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `i`. The value of `i` could be outside the pointer range, causing an out-of-range error.

Correction — Check Index Before Dereference

One possible correction is to validate the value of the index. If the index is inside the valid range, continue with the pointer dereferencing.

```
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(int i) {
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (i>0 && i<SIZE10) {
            c = pint[i];
        }
        free(pint);
    }
    return c;
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: `TAINTED_PTR_OFFSET`
Impact: Low
CWE ID: 122, 124, 129, 823

See Also

`Array access with tainted index` | `Use of tainted pointer` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Tainted division operand

Operands of division operation (`/`) come from an unsecure source

Description

This defect occurs when one or both integer operands in a division operation comes from nonsecure sources.

Risk

- If the numerator is the minimum possible value and the denominator is `-1`, your division operation overflows because the result cannot be represented by the current variable size.
- If the denominator is zero, your division operation fails possibly causing your program to crash.

These risks can be used to execute arbitrary code. This code is usually outside the scope of a program's implicit security policy.

Fix

Before performing the division, validate the values of the operands. Check for denominators of `0` or `-1`, and numerators of the minimum integer value.

Examples

Division of Function Arguments

```
extern void print_int(int);

int taintedintdivision(int usernum, int userden) {
    int r =  usernum/userden;
    print_int(r);
    return r;
}
```

This example function divides two argument variables, then prints and returns the result. The argument values are unknown and can cause division by zero or integer overflow.

Correction — Check Values

One possible correction is to check the values of the numerator and denominator before performing the division.

```
#include "limits.h"

extern void print_int(int);

int taintedintdivision(int usernum, int userden) {
    int r = 0;
    if (userden!=0 && !(usernum=INT_MIN && userden==-1)) {
        r = usernum/userden;
    }
    print_int(r);
```

```
    return r;
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: `TAINTED_INT_DIVISION`
Impact: Low
CWE ID: 189, 190, 369

See Also

`Integer division by zero` | `Float division by zero` | `Tainted modulo operand` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Tainted modulo operand

Operands of remainder operation (%) come from an unsecure source

Description

This defect occurs when one or both integer operands in a remainder operation (%) comes from nonsecure sources.

Risk

- If the second remainder operand is zero, your remainder operation fails, causing your program to crash.
- If the second remainder operand is `-1`, your remainder operation can overflow if the remainder operation is implemented based on the division operation that can overflow.
- If one of the operands is negative, the operation result is uncertain. For C89, the modulo operation is not standardized, so the result from negative operands is implementation-defined.

These risks can be exploited by attackers to gain access to your program or the target in general.

Fix

Before performing the modulo operation, validate the values of the operands. Check the second operand for values of `0` and `-1`. Check both operands for negative values.

Examples

Modulo with Function Arguments

```
extern void print_int(int);

int taintedintmod(int userden) {
    int rem =  128%userden;
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using an input argument. The argument is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

Correction — Check Operand Values

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
extern void print_int(int);

int taintedintmod(int userden) {
    int rem = 0;
    if (userden > 0) {
        rem = 128 % userden;
```

```
    }
    print_int(rem);
    return rem;
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: `TAINTED_INT_MOD`
Impact: Low
CWE ID: 369, 682

See Also

`Integer division by zero` | `Tainted division operand` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Tainted NULL or non-null-terminated string

Argument is from an unsecure source and may be NULL or not NULL-terminated

Description

This defect occurs when strings from nonsecure sources are used in string manipulation routines that implicitly dereference the string buffer, for instance, `strcpy` or `sprintf`.

Tainted NULL or non-null-terminated string raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

Note If you reference a string using the form `ptr[i]`, `*ptr`, or pointer arithmetic, Bug Finder raises a **Use of tainted pointer** defect instead. The **Tainted NULL or non-null-terminated string** defect is raised only when the pointer is used as a string.

Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

Examples

Getting String from Input Argument

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
```

```
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

Correction — Validate the Data

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

int sanitize_str(char* s) {
  int res = 0;
  if (s && (strlen(s) > 0)) { // TAINTED_STRING only flagged here
    // - string is not null
    // - string has a positive and limited size
    // - TAINTED_STRING on strlen used as a firewall
    res = 1;
  }
  return res;
}

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
      strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  if (sanitize_str(userstr))
    strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}
```

Correction — Validate the Data

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: `TAINTED_STRING`
Impact: Low
CWE ID: 120, 170, 476, 690, 822

See Also

`Tainted string format` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Tainted sign change conversion

Value from an unsecure source changes sign

Description

This defect occurs when values from nonsecure sources are converted, implicitly or explicitly, from signed to unsigned values.

For example, functions that use `size_t` as arguments implicitly convert the argument to an unsigned integer. Some functions that implicitly convert `size_t` are:

```
bcmp
memcpy
memmove
strncmp
strncpy
calloc
malloc
memalign
```

Risk

If you convert a small negative number to unsigned, the result is a large positive number. The large positive number can create security vulnerabilities. For example, if you use the unsigned value in:

- Memory size routines — causes allocating memory issues.
- String manipulation routines — causes buffer overflow.
- Loop boundaries — causes infinite loops.

Fix

To avoid converting unsigned negative values, check that the value being converted is within an acceptable range. For example, if the value represents a size, validate that the value is not negative and less than the maximum value size.

Examples

Set Memory Value with Size Argument

```
#include <stdlib.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedsignchange(int size) {
    char str[SIZE128] = "";
    if (size<SIZE128) {
        memset(str, 'c', size);
```

```
    }
}
```

In this example, a `char` buffer is created and filled using `memset`. The size argument to `memset` is an input argument to the function.

The call to `memset` implicitly converts `size` to unsigned integer. If `size` is a large negative number, the absolute value could be too large to represent as an integer, causing a buffer overflow.

Correction — Check Value of `size`

One possible correction is to check if `size` is inside the valid range. This correction checks if `size` is greater than zero and less than the buffer size before calling `memset`.

```
#include <stdlib.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void corrected_taintedsignchange(int size) {
    char str[SIZE128] = "";
    if (size>0 && size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: `TAINTED_SIGN_CHANGE`
Impact: Medium
CWE ID: 128, 131, 192, 194, 195

See Also

`Sign change integer conversion overflow` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Tainted string format

Input format argument is from an unsecure source

Description

This defect occurs when `printf`-style functions use a format specifier constructed from nonsecure sources.

Risk

If you use externally controlled elements to format a string, you can cause buffer overflow or data-representation problems. An attacker can use these string formatting elements to view the contents of a stack using `%x` or write to a stack using `%n`.

Fix

Pass a static string to format string functions. This fix ensures that an external actor cannot control the string.

Another possible fix is to allow only the expected number of arguments. If possible, use functions that do not support the vulnerable `%n` operator in format strings.

Examples

Get Elements from User Input

```
#include "stdio.h"

void taintedstringformat(char* userstr) {
    printf(userstr);
}
```

This example prints the input argument `userstr`. The string is unknown. If it contains elements such as `%`, `printf` can interpret `userstr` as a string format instead of a string, causing your program to crash.

Correction — Print as String

One possible correction is to print `userstr` explicitly as a string so that there is no ambiguity.

```
#include "stdio.h"

void taintedstringformat(char* userstr) {
    printf("%.20s", userstr);
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: `TAINTED_STRING_FORMAT`

Impact: Low
CWE ID: 134

See Also

`Tainted NULL or non-null-terminated string|Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Tainted size of variable length array

Size of the variable-length array (VLA) is from an unsecure source and may be zero, negative, or too large

Description

This defect occurs when the size of a variable length array (VLA) is obtained from a nonsecure source.

Risk

If an attacker changed the size of your VLA to an unexpected value, it can cause your program to crash or behave unexpectedly.

If the size is non-positive, the behavior of the VLA is undefined. Your program does not perform as expected.

If the size is unbounded, the VLA can cause memory exhaustion or stack overflow.

Fix

Validate your VLA size to make sure that it is positive and less than a maximum value.

Examples

Input Argument Used as Size of VLA

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedvlasize(int size) {

    int tabvla[size];
    int res = 0;
    for (int i=0 ; i<SIZE10 ; ++i) {
        tabvla[i] = i*i;
        res += tabvla[i];
    }
    return res;
}
```

In this example, a variable length array size is based on an input argument. Because this input argument value is not checked, the size may be negative or too large.

Correction — Check VLA Size

One possible correction is to check the size variable before creating the variable length array. This example checks if the size is larger than 10 and less than 100, before creating the VLA

```
enum {
    SIZE10  =  10,
```

```
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedvlasize(int size) {
    int res = 0;
    if (size>SIZE10 && size<SIZE100) {
        int tabvla[size];
        for (int i=0 ; i<SIZE10 ; ++i) {
            tabvla[i] = i*i;
            res += tabvla[i];
        }
    }
    return res;
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: `TAINTED_VLA_SIZE`
Impact: Medium
CWE ID: 128, 131, 770, 789

See Also

`Memory allocation with tainted size` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Use of externally controlled environment variable

Value of environment variable from an unsecure source

Description

This defect occurs when functions that add or change environment variables, such as `putenv` and `setenv`, obtain new environment variable values from nonsecure sources.

Risk

If the environment variable is tainted, an attacker can control your system settings. This control can disrupt an application or service in potentially malicious ways.

Fix

Before using the new environment variable, check its value to avoid giving control to external users.

Examples

Set Path in Environment

```
#define _XOPEN_SOURCE
#define _GNU_SOURCE
#include "stdlib.h"

void taintedenvvariable(char* path)
{
    putenv(path);
}
```

In this example, `putenv` changes an environment variable. The path `path` has not been checked to make sure that it is the intended path.

Correction — Sanitize Path

One possible correction is to sanitize the path, checking that it matches what you expect.

```
#define _XOPEN_SOURCE
#define _GNU_SOURCE
#define SIZE128 128
#include "stdlib.h"
#include "string.h"

/* Function to sanitize a string */
int sanitize_str(char* str, size_t n) {
    int res = 0;

    if (str && n > 0 && n < SIZE128) {
        /* string is not NULL, with size between 1 and max    */
        str[n-1] = '\0';  /* Add a null char at end of string */
        /* Tainted pointer detected above, used as "firewall" */
        res = 1;
    }
```

```
    return res;
}

void taintedenvvariable(char* path, size_t n)
{
    if (sanitize_str(path, n))
    {
        unsigned int n2 = strlen("PATH=")+strnlen(path, n);
        char *env_path = (char *)malloc(n2+1);
        if (env_path)
        {
            strcpy(env_path, "PATH=");
            strncat(env_path, path, n2);
            putenv(env_path);
        }
    }
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: `TAINTED_ENV_VARIABLE`
Impact: Medium
CWE ID: 15

See Also

`Execution of externally controlled command` | `Host change using externally controlled elements` | `Command executed from externally controlled path` | `Library loaded from externally controlled path` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Use of tainted pointer

Pointer from an unsecure source may be NULL or point to unknown memory

Description

This defect occurs when:

- Tainted NULL pointer — the pointer is not validated against NULL.
- Tainted size pointer — the size of the memory zone that a pointer points to is not validated.

Note On a single pointer, your code can have instances of **Use of tainted pointer**, **Pointer dereference with tainted offset**, and **Tainted NULL or non-null-terminated string**. Bug Finder raises only the first tainted pointer defect that it finds.

Risk

An attacker can give your program a pointer that points to unexpected memory locations. If the pointer is dereferenced to write, the attacker can:

- Modify the state variables of a critical program.
- Cause your program to crash.
- Execute unwanted code.

If the pointer is dereferenced to read, the attacker can:

- Read sensitive data.
- Cause your program to crash.
- Modify a program variable to an unexpected value.

Fix

Avoid use of pointers from external sources.

Alternatively, if you trust the external source, sanitize the pointer before dereference. In a separate sanitization function:

- Check that the pointer is not NULL.
- Check the size of the memory location (if possible). This second check validates whether the size of the data the pointer points to matches the size your program expects.

The defect still appears in the body of the sanitization function. However, if you use a sanitization function, instead of several occurrences, the defect appears only once. You can justify the defect and hide it in later reviews by using code annotations. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Function That Dereferences an External Pointer

```
void taintedptr(int* p, int i) {
    *p = i;
}
```

In this example, the pointer `*p` is passed as an argument, and the value is changed. The pointer can be null or point to unknown memory, which can be vulnerable.

Correction — Avoid Use of External Pointers

One possible correction is to avoid pointers from external sources.

```
#include <stdlib.h>
    int *taintedptr(int i) {
    /* Use heap memory allocated in the application */
    int *p = (int *)malloc(sizeof (int));
    if (p != NULL) { /* Check for success */
        *p = i;
    }
return p;
}
```

Correction — Check Pointer

Another possible correction is to sanitize the pointer before using it. This example uses a second function to check if the pointer is null and can be dereferenced.

The checker still shows a defect in the body of the sanitization function when you dereference the pointer. You can add code annotations to automatically justify this and other defects in the body of the sanitization function and hide them from the results list. See “Annotate Code and Hide Known or Acceptable Results”.

```
#include <stdlib.h>

int* sanitize_ptr(int* p) {
    int* res = NULL;
    int x;

    if (p != NULL) {
        x = *p; /* polyspace DEFECT:TAINTED_PTR,USELESS_WRITE [No action planned] "Check for dereference" */
        res = p;
    }
    return res;
}

void taintedptr(int *p, int i) {
    p = sanitize_ptr(p);
    if (p) {
        *p = i;
    }
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: `TAINTED_PTR`
Impact: Low

CWE ID: 690, 822

See Also

`Pointer dereference with tainted offset` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Concurrency Defects

Asynchronously cancellable thread

Calling thread might be cancelled in an unsafe state

Description

This defect occurs when you use `pthread_setcanceltype` with argument `PTHREAD_CANCEL_ASYNCHRONOUS` to set the cancellability type of a calling thread to asynchronous (or immediate) . An asynchronously cancellable thread can be cancelled at any time, usually immediately upon receiving a cancellation request.

Risk

The calling thread might be cancelled in an unsafe state that could result in a resources leak, a deadlock, a data race, data corruption, or unpredictable behavior.

Fix

Remove the call to `pthread_setcanceltype` with argument `PTHREAD_CANCEL_ASYNCHRONOUS` to use the default cancellability type `PTHREAD_CANCEL_DEFERRED` instead. With the default cancellability type, the thread defers cancellation requests until it calls a function that is a cancellation point.

Examples

Cancellability Type of Thread Set to Asynchronous

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static int fatal_error(void)
{
    exit(1);
}


volatile int a = 5;
volatile int b = 10;

pthread_mutex_t global_lock = PTHREAD_MUTEX_INITIALIZER;

void* swap_values_thread(void* dummy)
{
    int i;
    int c;
    int result;
    if ((result =
             pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &i)) != 0) {
        /* handle error */
        fatal_error();
    }
    while (1) {
        if ((result = pthread_mutex_lock(&global_lock)) != 0) {
            /* handle error */
            fatal_error();
        }
        c = b;
        b = a;
        a = c;
        if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
            /* handle error */
            fatal_error();
        }
```

```
    }
    return NULL;
}

int main(void)
{
    int result;
    pthread_t worker;

    if ((result = pthread_create(&worker, NULL, swap_values_thread, NULL)) != 0) {
        /* handle error */
        fatal_error();
    }

    /* Additional code */

    if ((result = pthread_cancel(worker)) != 0) {
        /* handle error */
        fatal_error();
    }


    if ((result = pthread_join(worker, 0)) != 0) {
        /* handle error */
        fatal_error();
    }

    if ((result = pthread_mutex_lock(&global_lock)) != 0) {
        /* handle error */
        fatal_error();
    }
    printf("a: %i | b: %i", a, b);
    if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
        /* handle error */
        fatal_error();
    }

    return 0;
}
```

In this example, the cancellability type of the `worker` thread is set to asynchronous. The mutex `global_lock` helps ensure that the `worker` and `main` threads do not access variables `a` and `b` at the same time. However, the `worker` thread might be cancelled while holding `global_lock`, and the `main` thread will never acquire `global_lock`, which results in a deadlock.

Correction — Use the Default Cancellability Type

One possible correction is to remove the call to `pthread_setcanceltype`. By default, the cancellability type of a new thread is set to `PTHREAD_CANCEL_DEFERRED`. The `worker` thread defers cancellation requests until it calls a function that is a cancellation point.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static int fatal_error(void)
{
    exit(1);
}


volatile int a = 5;
volatile int b = 10;

pthread_mutex_t global_lock = PTHREAD_MUTEX_INITIALIZER;

void* swap_values_thread(void* dummy)
{
    int i;
    int c;
    int result;
    while (1) {
        if ((result = pthread_mutex_lock(&global_lock)) != 0) {
            /* handle error */
            fatal_error();
        }
        c = b;
```

```
        b = a;
        a = c;
        if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
            /* handle error */
            fatal_error();
        }
    }
    return NULL;
}

int main(void)
{
    int result;
    pthread_t worker;

    if ((result = pthread_create(&worker, NULL, swap_values_thread, NULL)) != 0) {
        /* handle error */
        fatal_error();
    }

    /* Additional code */

    if ((result = pthread_cancel(worker)) != 0) {
        /* handle error */
        fatal_error();
    }


    if ((result = pthread_join(worker, 0)) != 0) {
        /* handle error */
        fatal_error();
    }

    if ((result = pthread_mutex_lock(&global_lock)) != 0) {
        /* handle error */
        fatal_error();
    }
    printf("a: %i | b: %i", a, b);
    if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
        /* handle error */
        fatal_error();
    }

    return 0;
}
```

Check Information

Group: Concurrency
Language: C | C++
Default: Off
Command-Line Syntax: ASYNCHRONOUSLY_CANCELLABLE_THREAD
Impact: Medium

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

External Websites

POS47-C

Introduced in R2020a

Atomic load and store sequence not atomic

Variable accessible between load and store operations

Description

This defect occurs when you use these functions to load, and then store an atomic variable.

- C functions:
  - `atomic_load()`
  - `atomic_load_explicit()`
  - `atomic_store()`
  - `atomic_store_explicit()`
- C++ functions:
  - `std::atomic_load()`
  - `std::atomic_load_explicit()`
  - `std::atomic_store()`
  - `std::atomic_store_explicit()`
  - `std::atomic::load()`
  - `std::atomic::store()`

A thread cannot interrupt an atomic load or an atomic store operation on a variable, but a thread can interrupt a store, and then load sequence.

Risk

A thread can modify a variable between the load and store operations, resulting in a data race condition.

Fix

To read, modify, and store a variable atomically, use a compound assignment operator such as `+=`, `atomic_compare_exchange()` or `atomic_fetch_*`-family functions.

Examples

Loading Then Storing an Atomic Variable

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void init_flag(void)
{
```

```
    atomic_init(&flag, false);
}

void toggle_flag(void)
{
    bool temp_flag = atomic_load(&flag);
    temp_flag = !temp_flag;
    atomic_store(&flag, temp_flag);
}

bool get_flag(void)
{
    return atomic_load(&flag);
}
```

In this example, variable `flag` of type `atomic_bool` is referenced twice inside the `toggle_flag()` function. The function loads the variable, negates its value, then stores the new value back to the variable. If two threads call `toggle_flag()`, the second thread can access `flag` between the load and store operations of the first thread. `flag` can end up in an incorrect state.

Correction — Use Compound Assignment to Modify Variable

One possible correction is to use a compound assignment operator to toggle the value of `flag`. The C standard defines the operation by using ^= as atomic.

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void toggle_flag(void)
{
    flag ^= 1;
}

bool get_flag(void)
{
    return flag;
}
```

Result Information

Group: Concurrency
Language: C | C++
Default: On
Command-Line Syntax: `ATOMIC_VAR_SEQUENCE_NOT_ATOMIC`
Impact: Medium

See Also

`Atomic variable accessed twice in an expression` | `Data race` | `Data race including atomic operations` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”

“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Atomic variable accessed twice in an expression

Variable can be modified between accesses

Description

This defect occurs when C atomic types or C++ `std::atomic` class variables appear twice in an expression and there are:

- Two atomic read operations on the variable.
- An atomic read and a distinct atomic write operation on the variable.

The C standard defines certain operations on atomic variables that are thread safe and do not cause data race conditions. Unlike individual operations, a pair of operations on the same atomic variable in an expression is not thread safe.

Risk

A thread can modify the atomic variable between the pair of atomic operations, which can result in a data race condition.

Fix

Do not reference an atomic variable twice in the same expression.

Examples

Referencing Atomic Variable Twice in an Expression

```
#include <stdatomic.h>

atomic_int n = ATOMIC_VAR_INIT(0);

int compute_sum(void)
{
    return n * (n + 1) / 2;
}
```

In this example, the global variable `n` is referenced twice in the return statement of `compute_sum()`. The value of `n` can change between the two distinct read operations. `compute_sum()` can return an incorrect value.

Correction — Pass Variable as Function Argument

One possible correction is to pass the variable as a function argument `n`. The variable is copied to memory and the read operations on the copy guarantee that `compute_sum()` returns a correct result. If you pass a variable of type `int` instead of type `atomic_int`, the correction is still valid.

```
#include <stdatomic.h>
```

```
int compute_sum(atomic_int n)
{
    return n * (n + 1) / 2;
}
```

Result Information

Group: Concurrency
Language: C | C++
Default: On
Command-Line Syntax: ATOMIC_VAR_ACCESS_TWICE
Impact: Medium

See Also

`Atomic load and store sequence not atomic` | `Data race` | `Data race including atomic operations` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Automatic or thread local variable escaping from a thread

Variable is passed from one thread to another without ensuring that variable stays alive through duration of latter thread

Description

This defect occurs when an automatic or thread local variable is passed by address from one thread to another thread without ensuring that the variable stays alive through the duration of the latter thread.

The defect checker applies to both C11 and POSIX threads.

Risk

An automatic or thread local variable is allocated on the stack at the beginning of a thread and its lifetime extends till the end of the thread. The variable is not guaranteed to be alive when a different thread accesses it.

For instance, consider the start function of a C11 thread with these lines:

```
int start_thread(thrd_t *tid) {
   int aVar = 0;
   if(thrd_success != thrd_create(tid, start_thread_child, &aVar) {
     ...
   }
}
```

The `thrd_create` function creates a child thread with start function `start_thread_child` and passes the address of the automatic variable `aVar` to this function. When this child thread accesses `aVar`, the parent thread might have completed execution and `aVar` is no longer on the stack. The access might result in reading unpredictable values.

Fix

When you pass a variable from one thread to another, make sure that the variable lifetime matches or exceeds the lifetime of both threads. You can achieve this synchronization in one of these ways:

- Declare the variable `static` so that it does not go out of stack when the current thread completes execution.
- Dynamically allocate the storage for the variable so that it is allocated on the heap instead of the stack and must be explicitly deallocated. Make sure that the deallocation happens after both threads complete execution.

These solutions require you to create a variable in nonlocal memory. Instead, you can use other solutions such as the `shared` keyword with OpenMP's threading interface that allows you to safely share local variables across threads.

Examples

Automatic or Thread-Local Variable Escaping Thread

```
#include <threads.h>
#include <stdio.h>

int create_child_thread(void *childVal) {
  int *res = (int *)childVal;
  printf("Result: %d\n", *res);
  return 0;
}

void create_parent_thread(thrd_t *tid, int *parentPtr) {
   if (thrd_success != thrd_create(tid, create_child_thread, parentPtr)) {
    /* Handle error */
  }
}

int main(void) {
  thrd_t tid;
  int parentVal = 1;

  create_parent_thread(&tid, &parentVal);


  if (thrd_success != thrd_join(tid, NULL)) {
    /* Handle error */
  }
  return 0;
}
```

In this example, the value `parentVal` is local to the parent thread that starts in `main` and continues into the function `create_parent_thread`. However, in the body of `create_parent_thread`, the address of this local variable is passed to a child thread (the thread with start routine `create_child_thread`). The parent thread might have completed execution and the variable `parentVal` might have gone out of scope when the child thread accesses this variable.

The same issue appears if the variable is declared as thread-local, for instance with the C11 keyword `_Thread_local` (or `thread_local`):

```
_Thread_local int parentVal = 1;
```

Correction – Use Static Variables

One possible correction is to declare the variable `parentVal` as `static` so that the variable is on the stack for the entire duration of the program.

```
#include <threads.h>
#include <stdio.h>

int create_child_thread(void *childVal) {
  int *res = (int *)childVal;
  printf("Result: %d\n", *res);
```

```
  return 0;
}

void create_parent_thread(thrd_t *tid, int *parentPtr) {
   if (thrd_success != thrd_create(tid, create_child_thread, parentPtr)) {
    /* Handle error */
  }
}

int main(void) {
  thrd_t tid;
  static int parentVal = 1;

  create_parent_thread(&tid, &parentVal);


  if (thrd_success != thrd_join(tid, NULL)) {
    /* Handle error */
  }
  return 0;
}
```

Correction – Use Dynamic Memory Allocation

Another possible correction is to dynamically allocate storage for variables to be shared across threads and explicitly free the storage after the threads complete execution.

```
#include <threads.h>
#include <stdio.h>

int create_child_thread(void *childVal) {
  int *res = (int *)childVal;
  printf("Result: %d\n", *res);
  return 0;
}

void create_parent_thread(thrd_t *tid, int *parentPtr) {
   if (thrd_success != thrd_create(tid, create_child_thread, parentPtr)) {
    /* Handle error */
  }
}

int main(void) {
  thrd_t tid;
  int parentPtr = (int*) malloc(sizeof(int));

  if(!parentPtr) {
      create_parent_thread(&tid, &parentVal);


      if (thrd_success != thrd_join(tid, NULL)) {
        /* Handle error */
      }
      free(parentPtr);
  }
  return 0;
}
```

Check Information

Group: Concurrency
Language: C | C++
Default: Off
Command-Line Syntax: `LOCAL_ADDR_ESCAPE_THREAD`
Impact: Medium

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Blocking operation while holding lock

Task performs lengthy operation while holding a lock

Description

This defect occurs when a task (thread) performs a potentially lengthy operation while holding a lock.

The checker considers calls to these functions as potentially lengthy:

- Functions that access a network such as `recv`
- System call functions such as `fork`, `pipe` and `system`
- Functions for I/O operations such as `getchar` and `scanf`
- File handling functions such as `fopen`, `remove` and `lstat`
- Directory manipulation functions such as `mkdir` and `rmdir`

The checker automatically detects certain primitives that hold and release a lock, for instance, `pthread_mutex_lock` and `pthread_mutex_unlock`. For the full list of primitives that are automatically detected, see "Auto-Detection of Thread Creation and Critical Section in Polyspace".

Risk

If a thread performs a lengthy operation when holding a lock, other threads that use the lock have to wait for the lock to be available. As a result, system performance can slow down or deadlocks can occur.

Fix

Perform the blocking operation before holding the lock or after releasing the lock.

Some functions detected by this checker can be called in a way that does not make them potentially lengthy. For instance, the function `recv` can be called with the parameter `O_NONBLOCK` which causes the call to fail if no message is available. When called with this parameter, `recv` does not wait for a message to become available.

Examples

Network I/O Operations with `recv` While Holding Lock

```
#include <pthread.h>
#include <sys/socket.h>

pthread_mutexattr_t attr;
pthread_mutex_t mutex;

void thread_foo(void *ptr) {
  unsigned int num;
  int result;
  int sock;

  /* sock is a connected TCP socket */
```

```
  if ((result = pthread_mutex_lock(&mutex)) != 0) {
    /* Handle Error */
  }

  if ((result = recv(sock, (void *)&num, sizeof(unsigned int), 0)) < 0) {
    /* Handle Error */
  }

  /* ... */

  if ((result = pthread_mutex_unlock(&mutex)) != 0) {
    /* Handle Error */
  }
}

int main() {
  pthread_t thread;
  int result;

  if ((result = pthread_mutexattr_settype(
      &attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_init(&mutex, &attr)) != 0) {
    /* Handle Error */
  }

  if (pthread_create(&thread, NULL,(void*(*)(void*))& thread_foo, NULL) != 0) {
    /* Handle Error */
  }

  /* ... */

  pthread_join(thread, NULL);

  if ((result = pthread_mutex_destroy(&mutex)) != 0) {
    /* Handle Error */
  }

  return 0;
}
```

In this example, in each thread created with `pthread_create`, the function `thread_foo` performs a network I/O operation with `recv` after acquiring a lock with `pthread_mutex_lock`. Other threads using the same lock variable `mutex` have to wait for the operation to complete and the lock to become available.

Correction — Perform Blocking Operation Before Acquiring Lock

One possible correction is to call `recv` before acquiring the lock.

```
#include <pthread.h>
#include <sys/socket.h>

pthread_mutexattr_t attr;
pthread_mutex_t mutex;
```

```
void thread_foo(void *ptr) {
  unsigned int num;
  int result;
  int sock;

  /* sock is a connected TCP socket */
  if ((result = recv(sock, (void *)&num, sizeof(unsigned int), 0)) < 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_lock(&mutex)) != 0) {
    /* Handle Error */
  }

    /* ... */

  if ((result = pthread_mutex_unlock(&mutex)) != 0) {
    /* Handle Error */
  }
}

int main() {
  pthread_t thread;
  int result;

  if ((result = pthread_mutexattr_settype(
      &attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_init(&mutex, &attr)) != 0) {
    /* Handle Error */
  }

  if (pthread_create(&thread, NULL,(void*(*)(void*))& thread_foo, NULL) != 0) {
    /* Handle Error */
  }

  /* ... */

  pthread_join(thread, NULL);

  if ((result = pthread_mutex_destroy(&mutex)) != 0) {
    /* Handle Error */
  }

  return 0;
}
```

Result Information

Group: Concurrency
Language: C | C++
Default: Off
Command-Line Syntax: BLOCKING_WHILE_LOCKED
Impact: Low

CWE ID: 667

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Multiple threads waiting on same condition variable

Using `cnd_signal` to wake up one of the threads might result in indefinite blocking

Description

This defect occurs when you use `cnd_signal` family functions to wake up one of at least two threads that are concurrently waiting on the same condition variable. For threads with the same priority level, `cnd_signal` family functions cause the thread scheduler to arbitrarily wake up on of the threads waiting on the condition variable that you signal with the `cnd_signal` family function.

The checkers flags the `cnd_signal` family function call. See the **Event** column in the **Results Details** pane to view the threads waiting on the same condition variable.

Risk

The thread that is woken up with a `cnd_signal` family function usually tests for a condition predicate. While the condition predicate is false, the thread waits again on the condition variable until it is woken up by another thread that signals the condition variable. It is possible that the program ends up in a state where no thread is available to signal the condition variable, which results in indefinite blocking.

Fix

Use `cnd_broadcast` family functions instead to wake all threads waiting on the condition variable, or use a different condition variable for each thread.

Examples

Use of `cnd_signal` to Wake Up One of Many Threads Waiting on Condition Variable

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <threads.h>

typedef int thrd_return_t;

static void fatal_error(void)
{
    exit(1);
}

enum { NTHREADS = 5 };

mtx_t mutex;
cnd_t cond;

thrd_return_t next_step(void* t)
{
    static size_t current_step = 0;
```

```
    size_t my_step = *(size_t*)t;

    if (thrd_success != mtx_lock(&mutex)) {
        /* Handle error */
        fatal_error();
    }

    printf("Thread %zu has the lock\n", my_step);
    while (current_step != my_step) {
        printf("Thread %zu is sleeping...\n", my_step);
        if (thrd_success !=
            cnd_wait(&cond, &mutex)) {
            /* Handle error */
            fatal_error();
        }
        printf("Thread %zu woke up\n", my_step);
    }
    /* Do processing ... */
    printf("Thread %zu is processing...\n", my_step);
    current_step++;

    /* Signal a waiting task */
    if (thrd_success !=
        cnd_signal(&cond)) {
        /* Handle error */
        fatal_error();
    }

    printf("Thread %zu is exiting...\n", my_step);

    if (thrd_success != mtx_unlock(&mutex)) {
        /* Handle error */
        fatal_error();
    }
    return (thrd_return_t)0;
}

int main(void)
{
    thrd_t threads[NTHREADS];
    size_t step[NTHREADS];

    if (thrd_success != mtx_init(&mutex, mtx_plain)) {
        /* Handle error */
        fatal_error();
    }
    if (thrd_success != cnd_init(&cond)) {
        /* Handle error */
        fatal_error();
    }
    /* Create threads */
    for (size_t i = 0; i < NTHREADS; ++i) {
        step[i] = i;
        if (thrd_success != thrd_create(&threads[i],
                                        next_step,
                                        &step[i])) {
            /* Handle error */
            fatal_error();
```

```
        }
    }
    /* Wait for all threads to complete */
    for (size_t i = NTHREADS; i != 0; --i) {
        if (thrd_success != thrd_join(threads[i - 1], NULL)) {
            /* Handle error */
            fatal_error();
        }
    }
    (void)mtx_destroy(&mutex);
    (void)cnd_destroy(&cond);
    return 0;
}
```

In this example, multiple threads are created and assigned step level. Each thread checks if its assigned step level matches the current step level (condition predicate). If the predicate is false, the thread goes back to waiting on the condition variable `cond`. The use of `cnd_signal` to signal the `cond` causes the thread scheduler to arbitrarily wake up one of the threads waiting on `cond`. This can result in indefinite blocking when the condition predicate of woken up thread is false and no other thread is available to signal `cond`.

Correction — Use `cnd_broadcast` to Wake up All the Threads

One possible correction is to use `cnd_broadcast` instead to signal `cond`. The function `cnd_signal` wakes up all the thread that are waiting on `cond`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <threads.h>

typedef int thrd_return_t;

static void fatal_error(void)
{
    exit(1);
}

enum { NTHREADS = 5 };

mtx_t mutex;
cnd_t cond;

thrd_return_t next_step(void* t)
{
    static size_t current_step = 0;
    size_t my_step = *(size_t*)t;

    if (thrd_success != mtx_lock(&mutex)) {
        /* Handle error */
        fatal_error();
    }

    printf("Thread %zu has the lock\n", my_step);
    while (current_step != my_step) {
        printf("Thread %zu is sleeping...\n", my_step);
        if (thrd_success !=
            cnd_wait(&cond, &mutex)) {
```

```
            /* Handle error */
            fatal_error();
        }
        printf("Thread %zu woke up\n", my_step);
    }
    /* Do processing ... */
    printf("Thread %zu is processing...\n", my_step);
    current_step++;

    /* Signal a waiting task */
    if (thrd_success !=
        cnd_broadcast(&cond)) {
        /* Handle error */
        fatal_error();
    }

    printf("Thread %zu is exiting...\n", my_step);

    if (thrd_success != mtx_unlock(&mutex)) {
        /* Handle error */
        fatal_error();
    }
    return (thrd_return_t)0;
}

int main_test_next_step(void)
{
    thrd_t threads[NTHREADS];
    size_t step[NTHREADS];

    if (thrd_success != mtx_init(&mutex, mtx_plain)) {
        /* Handle error */
        fatal_error();
    }
    if (thrd_success != cnd_init(&cond)) {
        /* Handle error */
        fatal_error();
    }
    /* Create threads */
    for (size_t i = 0; i < NTHREADS; ++i) {
        step[i] = i;
        if (thrd_success != thrd_create(&threads[i],
                                        next_step,
                                        &step[i])) {
            /* Handle error */
            fatal_error();
        }
    }
    /* Wait for all threads to complete */
    for (size_t i = NTHREADS; i != 0; --i) {
        if (thrd_success != thrd_join(threads[i - 1], NULL)) {
            /* Handle error */
            fatal_error();
        }
    }
    (void)mtx_destroy(&mutex);
    (void)cnd_destroy(&cond);
```

```
    return 0;
}
```

Result Information

Group: Concurrency
Language: C | C++
Default: Off
Command-Line Syntax: SIGNALED_COND_VAR_NOT_UNIQUE
Impact: Low

See Also

Data race | Deadlock | Find defects (-checkers) | Function that can spuriously fail not wrapped in loop | Function that can spuriously wake up not wrapped in loop | Missing lock | Missing unlock | Multiple mutexes with one conditional variable

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

External Websites

CON38-C

Introduced in R2020a

Data race

Multiple tasks perform unprotected nonatomic operations on shared variable

Description

This defect occurs when:

1. Multiple tasks perform unprotected operations on a shared variable.
2. At least one task performs a write operation.
3. At least one operation is nonatomic. For data race on both atomic and nonatomic operations, see `Data race including atomic operations`.

   See “Define Atomic Operations in Multitasking Code”.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

Risk

Data race can result in unpredictable values of the shared variable because you do not control the order of the operations in different tasks.

Data races between two write operations are more serious than data races between a write and read operation. Two write operations can interfere with each other and result in indeterminate values. To identify write-write conflicts, use the filters on the **Detail** column of the **Results List** pane. For these conflicts, the **Detail** column shows the additional line:

```
Variable value may be altered by write-write concurrent access.
```

See “Filter and Group Results”.

Fix

To fix this defect, protect the operations on the shared variable using critical sections, temporal exclusion or another means. See “Protections for Shared Variables in Multitasking Code”.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the  icon. For an example, see below.

Examples

Unprotected Operation on Global Variable from Multiple Tasks

```
int var;
```

```
void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
    var++;
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

Option	Specification	
Configure multitasking manually	☑	
Tasks (-entry-points)	task1 task2 task3	
Critical section details (-critical-section-begin -critical-section-end)	**Starting routine**	**Ending routine**
	begin_critical_section	end_critical_section

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points task1,task2,task3
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

In this example, the tasks `task1`, `task2`, and `task3` call the function `increment`. `increment` contains the operation `var++` that can involve multiple machine instructions including:

- Reading `var`.
- Writing an increased value to `var`.

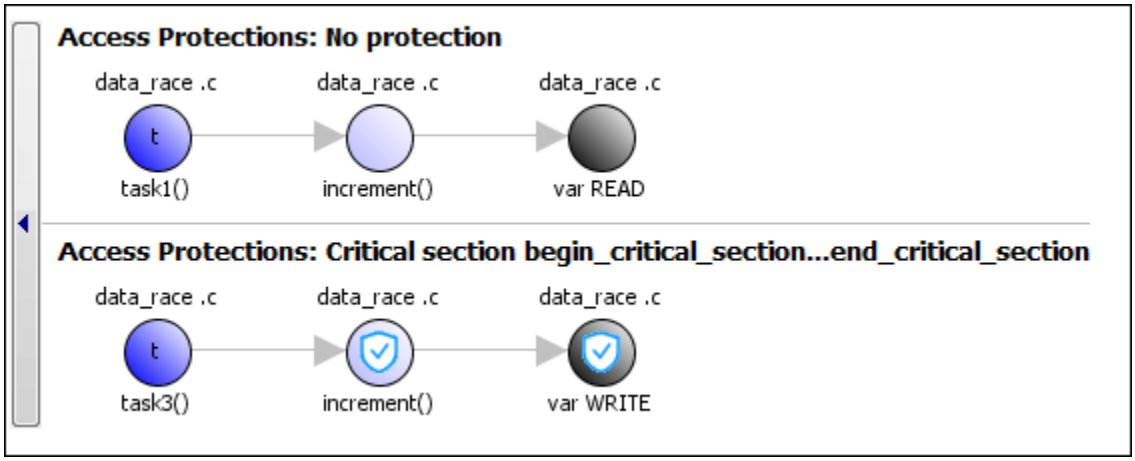
These machine instructions, when executed from `task1` and `task2`, can occur concurrently in an unpredictable sequence. For example, reading `var` from `task1` can occur either before or after writing to `var` from `task2`. Therefore the value of `var` can be unpredictable.

Though `task3` calls `increment` inside a critical section, other tasks do not use the same critical section. The operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

Therefore, the three tasks are operating on a shared variable without common protection. In your result details, you see each pair of conflicting function calls.

	Access	Access Protections	Task	File
	Read	No protection	task1()	data_race .c
	Write (Non atomic) Operation might involve multiple machine instructions	No protection	task2()	data_race .c
	Read	No protection	task1()	data_race .c
	Write (Non atomic) Operation might involve multiple machine instructions	**Critical section begin_critical_section...end_critical_section**	task3()	data_race .c
	Read	No protection	task2()	data_race .c
	Write (Non atomic) Operation might involve multiple machine instructions	**Critical section begin_critical_section...end_critical_section**	task3()	data_race .c

If you click the  icon, you see the function call sequence starting from the entry point to the read or write operation. You also see that the operation starting from `task3` is in a critical section. The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.



Correction — Place Operation in Critical Section

One possible correction is to place the operation in critical section. You can implement the critical section in multiple ways. For instance:

- You can place `var++` in a critical section. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The operation `var++` from the three tasks cannot interfere with each other.

  To implement the critical section, in the function `increment`, place the operation `var++` between calls to `begin_critical_section` and `end_critical_section`.

```
int var;

void begin_critical_section(void);
void end_critical_section(void);
```

```
void increment(void) {
      begin_critical_section();
      var++;
      end_critical_section();
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
      increment();
}
```

- You can place the call to `increment` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `increment` from the three tasks cannot interfere with each other.

  To implement the critical section, in each of the three tasks, call `increment` between calls to `begin_critical_section` and `end_critical_section`.

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
      var++;
}

void task1(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}

void task2(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

Option	Value
`Temporally exclusive tasks (-temporal-exclusions-file)`	`task1 task2 task3`

On the command-line, you can use the following:

```
 polyspace-bug-finder
     -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

Unprotected Operation in Threads Created with `pthread_create`

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;


void* increment_count(void* args)
{
    count = count + 1;
    return NULL;
}

void* set_count(void *args)
{
    long long c;
    c = count;
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);

    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

In this example, Bug Finder detects the creation of separate threads with `pthread_create`. The **Data race** defect is raised because the operation `count = count + 1` in the thread with id `thread_increment` conflicts with the operation `c = count` in the thread with id `thread_get`. The variable `count` is accessed in multiple threads without a common protection.

The two conflicting operations are nonatomic. The operation `c = count` is nonatomic on 32-bit targets. See “Define Atomic Operations in Multitasking Code”.

Correction — Protect Operations with `pthread_mutex_lock` and `pthread_mutex_unlock` Pair

To prevent concurrent access on the variable `count`, protect operations on `count` with a critical section. Use the functions `pthread_mutex_lock` and `pthread_mutex_unlock` to implement the critical section.

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;


void* increment_count(void* args)
{
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}

void* set_count(void *args)
{
    long long c;
    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);

    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

Check Information

Group: Concurrency
Language: C | C++
Default: On
Command-Line Syntax: `DATA_RACE`
Impact: High
CWE ID: 366, 413

See Also

`Configure multitasking manually` | `Critical section details (-critical-section-begin -critical-section-end)` | `Data race including atomic operations` | `Data race through standard library function call` | `Deadlock` | `Destruction of locked mutex` | `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)` | `Double lock` | `Double unlock` | `Find defects (-checkers)` | `Missing lock` | `Missing unlock` | `Target processor type (-target)` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics

“Analyze Multitasking Programs in Polyspace”
“Protections for Shared Variables in Multitasking Code”
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Define Atomic Operations in Multitasking Code”
“Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”

Introduced in R2014b

Data race including atomic operations

Multiple tasks perform unprotected operations on shared variable

Description

This defect occurs when:

1. Multiple tasks perform unprotected operations on a shared variable.
2. At least one task performs a write operation.

If you check for this defect, you can see data races on both atomic and non-atomic operations. To see data races on non-atomic operations alone, select `Data race`. Bug Finder considers an operation as atomic if it can be performed in one machine instruction. For instance, the operation:

```
int var = 0;
```

can be performed in one machine instruction on targets where the size of `int` is less than the word size on the target (or pointer size). See “Define Atomic Operations in Multitasking Code”. If you do not want to use this definition of atomic operations, turn on this checker.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Risk

Data race can result in unpredictable values of the shared variable because you do not control the order of the operations in different tasks.

Fix

To fix this defect, protect the operations on the shared variable using critical sections, temporal exclusion or another means. See “Protections for Shared Variables in Multitasking Code”.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the  icon. For an example, see below.

Examples

Unprotected Atomic Operation on Global Variable from Multiple Tasks

```
#include<stdio.h>

int var;

void begin_critical_section(void);
void end_critical_section(void);

void task1(void) {
```

```
    var = 1;
}

void task2(void) {
    int local_var;
    local_var = var;
    printf("%d", local_var);
}

void task3(void) {
    begin_critical_section();
    /* Operations in task3 */
    end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

Option	Specification	
Configure multitasking manually	☑	
Tasks (-entry-points)	task1 task2 task3	
Critical section details (-critical-section-begin -critical-section-end)	**Starting routine**	**Ending routine**
	begin_critical_section	end_critical_section

On the command-line, you can use the following:

```
polyspace-bug-finder
  -entry-points task1,task2,task3
  -critical-section-begin begin_critical_section:cs1
  -critical-section-end end_critical_section:cs1
```

In this example, the write operation `var=1;` in task `task1` executes concurrently with the read operation `local_var=var;` in task `task2`.

`task3` uses a critical section that can be reused for the other tasks.

Correction — Place Operations in Critical Section

One possible correction is to place these operations in the same critical section:

- `var=1;` in `task1`
- `local_var=var;` in `task2`

When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. Therefore, the two operations cannot execute concurrently.

To implement the critical section, reuse the already existing critical section in `task3`. Place the two operations between calls to `begin_critical_section` and `end_critical_section`.

```
#include<stdio.h>

int var;

void begin_critical_section();
void end_critical_section();

void task1(void) {
    begin_critical_section();
    var = 1;
    end_critical_section();
}

void task2(void) {
    int local_var;
    begin_critical_section();
    local_var = var;
    end_critical_section();
    printf("%d", local_var);
}

void task3(void) {
    begin_critical_section();
    /* Operations in task3 */
    end_critical_section();
}
```

Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks `task1` and `task2` temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

Option	Value
`Temporally exclusive tasks (-temporal-exclusions-file)`	`task1 task2`

On the command-line, use the following:

```
 polyspace-bug-finder
     -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2
```

Check Information

Group: Concurrency
Language: C | C++
Default: Off

Command-Line Syntax: `DATA_RACE_ALL`
Impact: Medium
CWE ID: 366, 413

See Also

`Configure multitasking manually` | `Critical section details (-critical-section-begin -critical-section-end)` | `Data race` | `Data race through standard library function call` | `Deadlock` | `Destruction of locked mutex` | `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)` | `Double lock` | `Double unlock` | `Find defects (-checkers)` | `Missing lock` | `Missing unlock` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics

“Configuring Polyspace Multitasking Analysis Manually”
“Protections for Shared Variables in Multitasking Code”
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Define Atomic Operations in Multitasking Code”

Introduced in R2014b

Data race through standard library function call

Multiple tasks make unprotected calls to thread-unsafe standard library function

Description

This defect occurs when:

1. Multiple tasks call the same standard library function.

   For instance, multiple tasks call the `strerror` function.
2. The calls are not protected using a common protection.

   For instance, the calls are not protected by the same critical section.

Functions flagged by this defect are not guaranteed to be reentrant. A function is reentrant if it can be interrupted and safely called again before its previous invocation completes execution. If a function is not reentrant, multiple tasks calling the function without protection can cause concurrency issues. For the list of functions that are flagged, see CON33-C: Avoid race conditions when using library functions.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

Risk

The functions flagged by this defect are nonreentrant because their implementations can use global or static variables. When multiple tasks call the function without protection, the function call from one task can interfere with the call from another task. The two invocations of the function can concurrently access the global or static variables and cause unpredictable results.

The calls can also cause more serious security vulnerabilities, such as abnormal termination, denial-of-service attack, and data integrity violations.

Fix

To fix this defect, do one of the following:

- Use a reentrant version of the standard library function if it exists.

  For instance, instead of `strerror()`, use `strerror_r()` or `strerror_s()`. For alternatives to functions flagged by this defect, see the documentation for CON33-C.
- Protect the function calls using common critical sections or temporal exclusion.

  See `Critical section details (-critical-section-begin -critical-section-end)` and `Temporally exclusive tasks (-temporal-exclusions-file)`.

  To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the  icon. For an example, see below.

Examples

Unprotected Call to Standard Library Function from Multiple Tasks

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno);
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

Option	Specification	
Configure multitasking manually	☑	
Tasks (-entry-points)	task1 task2 task3	
Critical section details (-critical-section-begin -critical-section-end)	**Starting routine**	**Ending routine**
	begin_critical_section	end_critical_section

On the command-line, you can use the following:

```
polyspace-bug-finder
  -entry-points task1,task2,task3
  -critical-section-begin begin_critical_section:cs1
  -critical-section-end end_critical_section:cs1
```

In this example, the tasks, `task1`, `task2` and `task3`, call the function `func`. `func` calls the nonreentrant standard library function, `strerror`.

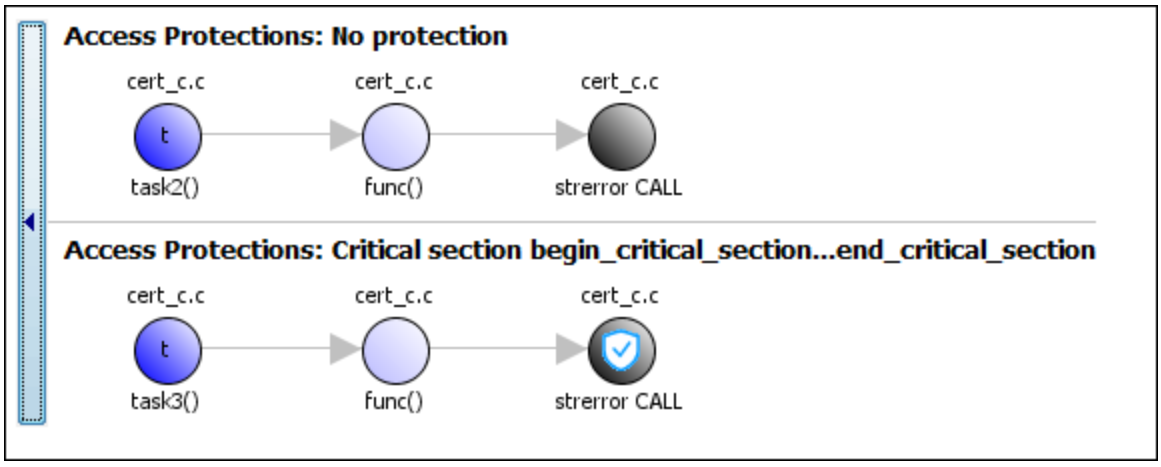
Though `task3` calls `func` inside a critical section, other tasks do not use the same critical section. Operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

These three tasks are calling a nonreentrant standard library function without common protection. In your result details, you see each pair of conflicting function calls.

! Data race through standard library function call (Impact: High)
Certain calls to function 'strerror' can interfere with each other and cause unpredictable results.
To avoid interference, calls to 'strerror' must be in the same critical section.

	Access	Access Protections	Task	File	Scope	Line
	Function call (Non atomic) Operation involves function call	No protection	task1()	data_race_std_lib.c	func()	14
	Function call (Non atomic) Operation involves function call	No protection	task2()	data_race_std_lib.c	func()	14
	Function call (Non atomic) Operation involves function call	No protection	task2()	data_race_std_lib.c	func()	14
	Function call (Non atomic) Operation involves function call	**Critical section begin_critical_section...end_critical_section**	task3()	data_race_std_lib.c	func()	14
	Function call (Non atomic) Operation involves function call	No protection	task1()	data_race_std_lib.c	func()	14
	Function call (Non atomic) Operation involves function call	**Critical section begin_critical_section...end_critical_section**	task3()	data_race_std_lib.c	func()	14

If you click the  icon, you see the function call sequence starting from the entry point to the standard library function call. You also see that the call starting from `task3` is in a critical section. The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.



Correction — Use Reentrant Version of Standard Library Function

One possible correction is to use a reentrant version of the standard library function `strerror`. You can use the POSIX version `strerror_r` which has the same functionality but also guarantees thread-safety.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);
enum { BUFFERSIZE = 64 };

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char errmsg[BUFFERSIZE];
    if (strerror_r(errno, errmsg, BUFFERSIZE) != 0) {
      /* Handle error */
    }
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

Correction — Place Function Call in Critical Section

One possible correction is to place the call to `strerror` in critical section. You can implement the critical section in multiple ways.

For instance, you can place the call to the intermediate function `func` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `func` and therefore the calls to `strerror` from the three tasks cannot interfere with each other.

To implement the critical section, in each of the three tasks, call `func` between calls to `begin_critical_section` and `end_critical_section`.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno);
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    begin_critical_section();
    func(fptr1);
    end_critical_section();
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     begin_critical_section();
     func(fptr2);
     end_critical_section();
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

Option	Value
Temporally exclusive tasks (-temporal-exclusions-file)	task1 task2 task3

On the command-line, you can use the following:

```
 polyspace-bug-finder
     -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

Result Information

Group: Concurrency
Language: C | C++
Default: On
Command-Line Syntax: `DATA_RACE_STD_LIB`
Impact: High
CWE ID: 366, 413

See Also

`Configure multitasking manually` | `Critical section details (-critical-section-begin -critical-section-end)` | `Data race` | `Data race including atomic operations` | `Destruction of locked mutex` | `Double lock` | `Double unlock` | `Find defects (-checkers)` | `Missing lock` | `Missing unlock` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics

“Analyze Multitasking Programs in Polyspace”
“Protections for Shared Variables in Multitasking Code”
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2016b

Deadlock

Call sequence to lock functions cause two tasks to block each other

Description

This defect occurs when multiple tasks are stuck in their critical sections (CS) because:

- Each CS waits for another CS to end.
- The critical sections (CS) form a closed cycle. For example:
  - CS #1 waits for CS #2 to end, and CS #2 waits for CS #1 to end.
  - CS #1 waits for CS #2 to end, CS #2 waits for CS #3 to end and CS #3 waits for CS #1 to end.

Polyspace expects critical sections of code to follow a specific format. A critical section lies between a call to a lock function and a call to an unlock function. When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait until `my_task` calls the corresponding unlock function. Both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Risk

Each task waits for a critical section in another task to end and is unable to proceed. The program can freeze indefinitely.

Fix

The fix depends on the root cause of the defect. You can try to break the cyclic order between the tasks in one of these ways:

- Write down all critical sections involved in the deadlock in a certain sequence. Whenever you call the lock functions of the critical sections within a task, respect the order in that sequence. See an example below.
- If one of the critical sections involved in a deadlock occurs in an interrupt, try to disable all interrupts during critical sections in all tasks. See `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Reviewing this defect is an opportunity to check if all operations in your critical section are really meant to be executed as an atomic block. It is a good practice to keep critical sections at a bare minimum.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Deadlock with Two Tasks

```
void task1(void);
void task2(void);

int var;
void perform_task_cycle(void) {
 var++;
}

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_2();
    begin_critical_section_1();
    perform_task_cycle();
    end_critical_section_1();
    end_critical_section_2();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Specification	
Configure multitasking manually	☑	
Tasks (-entry-points)	task1 task2	
Critical section details (-critical-section-begin -critical-section-end)	**Starting routine**	**Ending routine**
	begin_critical_section_1	end_critical_section_1
	begin_critical_section_2	end_critical_section_2

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. `task1` calls `begin_critical_section_1`.
2. `task2` calls `begin_critical_section_2`.
3. `task1` reaches the instruction `begin_critical_section_2();`. Since `task2` has already called `begin_critical_section_2`, `task1` waits for `task2` to call `end_critical_section_2`.
4. `task2` reaches the instruction `begin_critical_section_1();`. Since `task1` has already called `begin_critical_section_1`, `task2` waits for `task1` to call `end_critical_section_1`.

Correction-Follow Same Locking Sequence in Both Tasks

One possible correction is to follow the same sequence of calls to lock and unlock functions in both `task1` and `task2`.

```
void task1(void);
void task2(void);
void perform_task_cycle(void);

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}
```

Deadlock with More Than Two Tasks

```
int var;
void performTaskCycle() {
```

```
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);


void unlock1(void);
void unlock2(void);
void unlock3(void);

void task1() {
 while(1) {
    lock1();
    lock2();
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock3();
    lock1();
    performTaskCycle();
    unlock1();
    unlock3();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Specification	
Configure multitasking manually	☑	

Entry points	`task1`
	`task2`
	`task3`	

Critical section details	**Starting routine**	**Ending routine**
	`lock1`	`unlock1`

Option	Specification	
	`lock2`	`unlock2`
	`lock3`	`unlock3`

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. `task1` calls `lock1`.
2. `task2` calls `lock2`.
3. `task3` calls `lock3`.
4. `task1` reaches the instruction `lock2();`. Since `task2` has already called `lock2`, `task1` waits for call to `unlock2`.
5. `task2` reaches the instruction `lock3();`. Since `task3` has already called `lock3`, `task2` waits for call to `unlock3`.
6. `task3` reaches the instruction `lock1();`. Since `task1` has already called `lock1`, `task3` waits for call to `unlock1`.

Correction — Break Cyclic Order

To break the cyclic order between critical sections, note every lock function in your code in a certain sequence, for example:

1. `lock1`
2. `lock2`
3. `lock3`

If you use more than one lock function in a task, use them in the order in which they appear in the sequence. For example, you can use `lock1` followed by `lock2` but not `lock2` followed by `lock1`.

```
int var;
void performTaskCycle() {
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);

void unlock1(void);
void unlock2(void);
void unlock3(void);

void task1() {
 while(1) {
    lock1();
    lock2();
    performTaskCycle();
    unlock2();
    unlock1();
 }
}
```

```
void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock1();
    lock3();
    performTaskCycle();
    unlock3();
    unlock1();
 }
}
```

Check Information

Group: Concurrency
Language: C | C++
Default: On
Command-Line Syntax: DEADLOCK
Impact: High
CWE ID: 833

See Also

`Configure multitasking manually` | `Critical section details (-critical-section-begin -critical-section-end)` | `Data race` | `Data race including atomic operations` | `Data race through standard library function call` | `Destruction of locked mutex` | `Double lock` | `Double unlock` | `Find defects (-checkers)` | `Missing lock` | `Missing unlock` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics

“Analyze Multitasking Programs in Polyspace”
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”

Introduced in R2014b

Destruction of locked mutex

Task tries to destroy a mutex in the locked state

Description

This defect occurs when a task destroys a mutex after it is locked (and before it is unlocked). The locking and destruction can happen in the same task or different tasks.

Risk

A mutex is locked to protect shared variables from concurrent access. If a mutex is destroyed in the locked state, the protection does not apply.

Fix

To fix this defect, destroy the mutex only after you unlock it. It is a good design practice to:

- Initialize a mutex *before* creating the threads where you use the mutex.
- Destroy a mutex *after* joining the threads that you created.

On the **Result Details** pane, you see two events, the locking and destruction of the mutex, and the tasks that initiated the events. To navigate to the corresponding line in your source code, click the event.

Examples

Locking and Destruction in Different Tasks

```
#include <pthread.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;
pthread_mutex_t lock3;

void t0 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
  pthread_mutex_lock (&lock3);
  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
  pthread_mutex_unlock (&lock3);
}

void t1 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
  pthread_mutex_destroy (&lock3);
  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}
```

In this example, after task `t0` locks the mutex `lock3`, task `t1` can destroy it. The destruction occurs if the following events happen in sequence:

1. `t0` acquires `lock3`.
2. `t0` releases `lock2`.
3. `t0` releases `lock1`.
4. `t1` acquires the lock `lock1` released by `t0`.
5. `t1` acquires the lock `lock2` released by `t0`.
6. `t1` destroys `lock3`.

For simplicity, this example uses a mix of automatic and manual concurrency detection. The tasks `t0` and `t1` are manually specified as entry points by using the option `Tasks (-entry-points)`. The critical sections are implemented through primitives `pthread_mutex_lock` and `pthread_mutex_unlock` that the software detects automatically. In practice, for entry point specification (thread creation), you will use primitives such as `pthread_create`. The next example shows how the defect can appear when you use `pthread_create`.

Correction — Place Lock-Unlock Pair Together in Same Critical Section as Destruction

The locking and destruction of `lock3` occurs inside the critical section imposed by `lock1` and `lock2`, but the unlocking occurs outside. One possible correction is to place the lock-unlock pair in the same critical section as the destruction of the mutex. Use one of these critical sections:

- Critical section imposed by `lock1` alone.
- Critical section imposed by `lock1` and `lock2`.

In this corrected code, the lock-unlock pair and the destruction is placed in the critical section imposed by `lock1` and `lock2`. When `t0` acquires `lock1` and `lock2`, `t1` has to wait for their release before it executes the instruction `pthread_mutex_destroy (&lock3);`. Therefore, `t1` cannot destroy mutex `lock3` in the locked state.

```
#include <pthread.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;
pthread_mutex_t lock3;

void t0 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);

  pthread_mutex_lock (&lock3);
  pthread_mutex_unlock (&lock3);

  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}

void t1 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
```

```
  pthread_mutex_destroy (&lock3);

  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}
```

Locking and Destruction in Start Routine of Thread

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 4
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
void atomic_operation(void);

void *do_create(void *arg) {
    /* Creation thread */
    pthread_mutex_init(&lock, NULL);
    pthread_exit((void*) 0);
}

void *do_work(void *arg) {
    /* Worker thread */
    pthread_mutex_lock (&lock);
    atomic_operation();
    pthread_mutex_unlock (&lock);
    pthread_exit((void*) 0);
}

void *do_destroy(void *arg) {
    /* Destruction thread */
    pthread_mutex_destroy(&lock);
    pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Thread that initializes mutex */
   pthread_create(&callThd[0], &attr, do_create, NULL);

   /* Threads that use mutex for atomic operation*/
   for(i=0; i<NUMTHREADS-1; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   /* Thread that destroys mutex */
   pthread_create(&callThd[NUMTHREADS -1], &attr, do_destroy, NULL);
```

```
   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   pthread_exit(NULL);
}
```

In this example, four threads are created. The threads are assigned different actions.

- The first thread `callThd[0]` initializes the mutex `lock`.
- The second and third threads, `callThd[1]` and `callThd[2]`, perform an atomic operation protected by the mutex `lock`.
- The fourth thread `callThd[3]` destroys the mutex `lock`.

The threads can interrupt each other. Therefore, immediately after the second or third thread locks the mutex, the fourth thread can destroy it.

Correction — Initialize and Destroy Mutex Outside Start Routine

One possible correction is to initialize and destroy the mutex in the `main` function outside the start routine of the threads. The threads perform only the atomic operation. You need two fewer threads because the mutex initialization and destruction threads are not required.

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 2
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
void atomic_operation(void);

void *do_work(void *arg) {
   pthread_mutex_lock (&lock);
   atomic_operation();
   pthread_mutex_unlock (&lock);
   pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Initialize mutex */
   pthread_mutex_init(&lock, NULL);

   for(i=0; i<NUMTHREADS; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }
```

```
   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* Destroy mutex */
   pthread_mutex_destroy(&lock);

   pthread_exit(NULL);
}
```

Correction — Use A Second Mutex To Protect Lock-Unlock Pair and Destruction

Another possible correction is to use a second mutex and protect the lock-unlock pair from the destruction. This corrected code uses the mutex `lock2` to achieve this protection. The second mutex is initialized in the `main` function outside the start routine of the threads.

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 4
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
pthread_mutex_t lock2;
void atomic_operation(void);

void *do_create(void *arg) {
    /* Creation thread */
    pthread_mutex_init(&lock, NULL);
    pthread_exit((void*) 0);
}

void *do_work(void *arg) {
    /* Worker thread */
    pthread_mutex_lock (&lock2);
    pthread_mutex_lock (&lock);
    atomic_operation();
    pthread_mutex_unlock (&lock);
    pthread_mutex_unlock (&lock2);
    pthread_exit((void*) 0);
}

void *do_destroy(void *arg) {
    /* Destruction thread */
    pthread_mutex_lock (&lock2);
    pthread_mutex_destroy(&lock);
    pthread_mutex_unlock (&lock2);
    pthread_exit((void*) 0);
}


int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;
```

```
   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Initialize second mutex */
   pthread_mutex_init(&lock2, NULL);

   /* Thread that initializes first mutex */
   pthread_create(&callThd[0], &attr, do_create, NULL);

   /* Threads that use first mutex for atomic operation */
   /* The threads use second mutex to protect first from destruction in locked state*/
   for(i=0; i<NUMTHREADS-1; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   /* Thread that destroys first mutex */
   /* The thread uses the second mutex to prevent destruction of locked mutex */
   pthread_create(&callThd[NUMTHREADS -1], &attr, do_destroy, NULL);


   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* Destroy second mutex */
   pthread_mutex_destroy(&lock2);

   pthread_exit(NULL);
}
```

Result Information

Group: Concurrency
Language: C | C++
Default: Off
Command-Line Syntax: DESTROY_LOCKED
Impact: Medium
CWE ID: 667, 826

See Also

`Configure multitasking manually` | `Data race` | `Data race including atomic operations` | `Data race through standard library function call` | `Deadlock` | `Double lock` | `Double unlock` | `Find defects (-checkers)` | `Missing lock` | `Missing unlock` | `Target processor type (-target)` | `Tasks (-entry-points)`

Topics

“Analyze Multitasking Programs in Polyspace”
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2016b

Double lock

Lock function is called twice in a task without an intermediate call to unlock function

Description

This defect occurs when:

- A task calls a lock function `my_lock`.
- The task calls `my_lock` again before calling the corresponding unlock function.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task `task1` calls a lock function `lock`, other tasks calling `lock` must wait until `task` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Risk

A call to a lock function begins a critical section so that other tasks have to wait to enter the same critical section. If the same lock function is called again within the critical section, the task blocks itself.

Fix

The fix depends on the root cause of the defect. A double lock defect often indicates a coding error. Perhaps you omitted the call to an unlock function to end a previous critical section and started the next critical section. Perhaps you wanted to use a different lock function for the second critical section.

Identify each critical section of code, that is, the section that you want to be executed as an atomic block. Call a lock function at the beginning of the section. Within the critical section, make sure that you do not call the lock function again. At the end of the section, call the unlock function that corresponds to the lock function.

See examples of fixes below. To avoid the issue, you can follow the practice of calling the lock and unlock functions in the same module at the same level of abstraction. For instance, in this example, `func` calls the lock and unlock function at the same level but `func2` does not.

```
void func() {
  my_lock();
  {
    ...
  }
  my_unlock();
}

void func2() {
  {
   my_lock();
   ...
```

```
  }
  my_unlock();
}
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Double Lock

```
int global_var;

void lock(void);
void unlock(void);

void task1(void)
{
    lock();
    global_var += 1;
    lock();
    global_var += 1;
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Value	
Configure multitasking manually	☑	
Tasks (-entry-points)	task1 task2	
Critical section details (-critical-section-begin -critical-section-end)	**Starting routine**	**Ending routine**
	lock	unlock

On the command-line, you can use the following:

```
polyspace-bug-finder
  -entry-points task1,task2
  -critical-section-begin lock:cs1
  -critical-section-end unlock:cs1
```

`task1` enters a critical section through the call `lock();`. `task1` calls `lock` again before it leaves the critical section through the call `unlock();`.

Correction — Remove First Lock

If you want the first `global_var+=1;` to be outside the critical section, one possible correction is to remove the first call to `lock`. However, if other tasks are using `global_var`, this code can produce a `Data race` error.

```
int global_var;

void lock(void);
void unlock(void);

void task1(void)
{
    global_var += 1;
    lock();
    global_var += 1;
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

Correction — Remove Second Lock

If you want the first `global_var+=1;` to be inside the critical section, one possible correction is to remove the second call to `lock`.

```
int global_var;

void lock(void);
void unlock(void);

void task1(void)
{
    lock();
    global_var += 1;
    global_var += 1;
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

Correction — Add Another Unlock

If you want the second `global_var+=1;` to be inside a critical section, another possible correction is to add another call to `unlock`.

```
int global_var;

void lock(void);
void unlock(void);

void task1(void)
{
    lock();
    global_var += 1;
    unlock();
    lock();
    global_var += 1;
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

Double Lock with Function Call

```
int global_var;

void lock(void);
void unlock(void);

void performOperation(void) {
  lock();
  global_var++;
}

void task1(void)
{
    lock();
    global_var += 1;
    performOperation();
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
```

```
    unlock();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Specification	
Configure multitasking manually	☑	
Tasks (-entry-points)	task1 task2	
Critical section details (-critical-section-begin -critical-section-end)	**Starting routine**	**Ending routine**
	lock	unlock

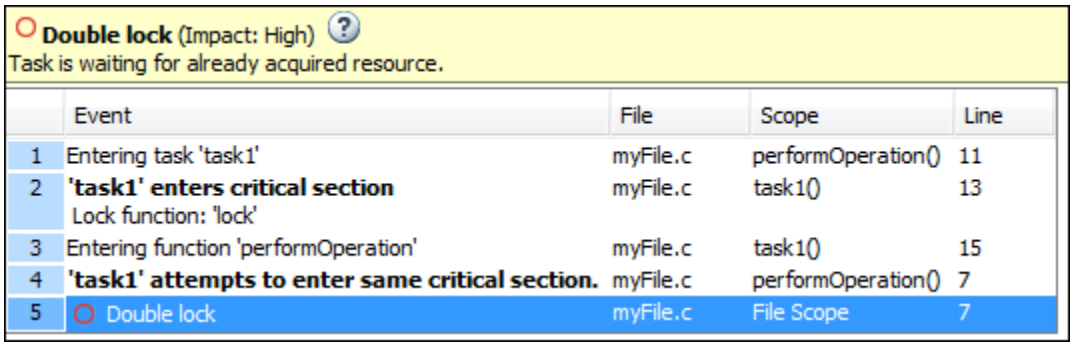
On the command-line, you can use the following:

```
polyspace-bug-finder
  -entry-points task1,task2
  -critical-section-begin lock:cs1
  -critical-section-end unlock:cs1
```

`task1` enters a critical section through the call `lock();`. `task1` calls the function `performOperation`. In `performOperation`, `lock` is called again even though `task1` has not left the critical section through the call `unlock();`.

In the result details for the defect, you see the sequence of instructions leading to the defect. For instance, you see that following the first entry into the critical section, the execution path:

- Enters function `performOperation`.
- Inside `performOperation`, attempts to enter the same critical section once again.



You can click each event to navigate to the corresponding line in the source code.

Correction — Remove Second Lock

One possible correction is to remove the call to `lock` in `task1`.

```
int global_var;
```

```
void lock(void);
void unlock(void);

void performOperation(void) {
  global_var++;
}

void task1(void)
{
    lock();
    global_var += 1;
    performOperation();
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

Check Information

Group: Concurrency
Language: C | C++
Default: On
Command-Line Syntax: DOUBLE_LOCK
Impact: High
CWE ID: 764

See Also

`Configure multitasking manually` | `Critical section details (-critical-section-begin -critical-section-end)` | `Data race` | `Data race including atomic operations` | `Data race through standard library function call` | `Deadlock` | `Destruction of locked mutex` | `Double unlock` | `Find defects (-checkers)` | `Missing lock` | `Missing unlock` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics

“Analyze Multitasking Programs in Polyspace”
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”

Introduced in R2014b

Double unlock

Unlock function is called twice in a task without an intermediate call to lock function

Description

This defect occurs when:

- A task calls a lock function `my_lock`.
- The task calls the corresponding unlock function `my_unlock`.
- The task calls `my_unlock` again. The task does not call `my_lock` a second time between the two calls to `my_unlock`.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task `task1` calls a lock function `my_lock`, other tasks calling `my_lock` must wait until `task1` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Risk

A double unlock defect can indicate a coding error. Perhaps you wanted to call a different unlock function to end a different critical section. Perhaps you called the unlock function prematurely the first time and only the second call indicates the end of the critical section.

Fix

The fix depends on the root cause of the defect.

Identify each critical section of code, that is, the section that you want to be executed as an atomic block. Call a lock function at the beginning of the section. Only at the end of the section, call the unlock function that corresponds to the lock function. Remove any other redundant call to the unlock function.

See examples of fixes below. To avoid the issue, you can follow the practice of calling the lock and unlock functions in the same module at the same level of abstraction. For instance, in this example, `func` calls the lock and unlock function at the same level but `func2` does not.

```
void func() {
  my_lock();
  {
    ...
  }
  my_unlock();
}

void func2() {
  {
   my_lock();
   ...
  }
```

```
  my_unlock();
}
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Double Unlock

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void task1(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}

void task2(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Value	
Configure multitasking manually	☑	
Tasks (-entry-points)	task1 task2	
Critical section details (-critical-section-begin -critical-section-end)	**Starting routine**	**Ending routine**
	BEGIN_CRITICAL_SECTION	END_CRITICAL_SECTION

On the command-line, you can use the following:

```
polyspace-bug-finder
  -entry-points task1,task2
  -critical-section-begin BEGIN_CRITICAL_SECTION:cs1
  -critical-section-end END_CRITICAL_SECTION:cs1
```

`task1` enters a critical section through the call `BEGIN_CRITICAL_SECTION();`. `task1` leaves the critical section through the call `END_CRITICAL_SECTION();`. `task1` calls `END_CRITICAL_SECTION` again without an intermediate call to `BEGIN_CRITICAL_SECTION`.

Correction — Remove Second Unlock

If you want the second `global_var+=1;` to be outside the critical section, one possible correction is to remove the second call to `END_CRITICAL_SECTION`. However, if other tasks are using `global_var`, this code can produce a `Data race` error.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void task1(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
    global_var += 1;
}

void task2(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

Correction — Remove First Unlock

If you want the second `global_var+=1;` to be inside the critical section, one possible correction is to remove the first call to `END_CRITICAL_SECTION`.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void task1(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    global_var += 1;
    END_CRITICAL_SECTION();
}

void task2(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
```

```
    END_CRITICAL_SECTION();
}
```

Correction — Add Another Lock

If you want the second `global_var+=1;` to be inside a critical section, another possible correction is to add another call to `BEGIN_CRITICAL_SECTION`.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void task1(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}

void task2(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

Check Information

Group: Concurrency
Language: C | C++
Default: On
Command-Line Syntax: `DOUBLE_UNLOCK`
Impact: High
CWE ID: 765

See Also

`Configure multitasking manually` | `Critical section details (-critical-section-begin -critical-section-end)` | `Data race` | `Data race including atomic operations` | `Data race through standard library function call` | `Deadlock` | `Destruction of locked mutex` | `Double lock` | `Find defects (-checkers)` | `Missing lock` | `Missing unlock` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics

“Analyze Multitasking Programs in Polyspace”
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”

Introduced in R2014b

Function that can spuriously fail not wrapped in loop

Loop checks failure condition after possible spurious failure

Description

This defect occurs when the following atomic compare and exchange functions that can fail spuriously are called from outside a loop.

- C atomic functions:
  - `atomic_compare_exchange_weak()`
  - `atomic_compare_exchange_weak_explicit()`
- C++ atomic functions:
  - `std::atomic<T>::compare_exchange_weak(T* expected, T desired)`
  - `std::atomic<T>::compare_exchange_weak_explicit(T* expected, T desired, std::memory_order succ, std::memory_order fail)`
  - `std::atomic_compare_exchange_weak(std::atomic<T>* obj, T* expected, T desired)`
  - `std::atomic_compare_exchange_weak_explicit(volatile std::atomic<T>* obj, T* expected, T desired, std::memory_order succ, std::memory_order fail)`

The functions compare the memory contents of the object representations pointed to by `obj` and `expected`. The comparison can spuriously return false even if the memory contents are equal. This spurious failure makes the functions faster on some platforms.

Risk

An atomic compare and exchange function that spuriously fails can cause unexpected results and unexpected control flow.

Fix

Wrap atomic compare and exchange functions that can spuriously fail in a loop. The loop checks the failure condition after a possible spurious failure.

Examples

`atomic_compare_exchange_weak()` Not Wrapped in Loop

```
#include <stdatomic.h>

extern void reset_count(void);
atomic_int count = ATOMIC_VAR_INIT(0);

void increment_count(void)
{
    int old_count = atomic_load(&count);
    int new_count;
    new_count = old_count + 1;
    if (!atomic_compare_exchange_weak(&count, &old_count, new_count))
```

```
        reset_count();

}
```

In this example, `increment_count()` uses `atomic_compare_exchange_weak()` to compare `count` and `old_count`. If the counts are equal, `count` is incremented to `new_count`. If they are not equal, the count is reset. When `atomic_compare_exchange_weak()` fails spuriously, the count is reset unnecessarily.

Correction — Wrap `atomic_compare_exchange_weak()` in a `while` Loop

One possible correction is to wrap the call to `atomic_compare_exchange_weak()` in a `while` loop. The loop checks the failure condition after a possible spurious failure.

```
#include <stdatomic.h>

extern void reset_count(void);
atomic_int count = ATOMIC_VAR_INIT(0);

void increment_count(void)
{
    int old_count = atomic_load(&count);
    int new_count;
    new_count = old_count + 1;

    do {
        reset_count();

    } while (!atomic_compare_exchange_weak(&count, &old_count, new_count));

}
```

Result Information

Group: Concurrency
Language: C | C++
Default: Off
Command-Line Syntax: `SPURIOUS_FAILURE_NOT_WRAPPED_IN_LOOP`
Impact: Low

See Also

`Find defects (-checkers)` | `Function that can spuriously wake up not wrapped in loop` | `Returned value of a sensitive function not checked`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Function that can spuriously wake up not wrapped in loop

Loop checks wake-up condition after possible spurious wake-up

Description

This defect occurs when the following wait-on-condition functions are called from outside a loop:

- C functions:
  - `cnd_wait()`
  - `cnd_timedwait()`
- POSIX functions:
  - `pthread_cond_wait()`
  - `pthread_cond_timedwait()`
- C++ `std::condition_variable` and `std::condition_variable_any` class member functions:
  - `wait()`
  - `wait_until()`
  - `wait_for()`

Wait-on-condition functions pause the execution of the calling thread when a specified condition is met. The thread wakes up and resumes once another thread notifies it with `cnd_broadcast()` or an equivalent function. The wake-up notification can be spurious or malicious.

Risk

If a thread receives a spurious wake-up notification and the condition of the wait-on-condition function is not checked, the thread can wake up prematurely. The wake-up can cause unexpected control flow, indefinite blocking of other threads, or denial of service.

Fix

Wrap wait-on-condition functions that can wake up spuriously in a loop. The loop checks the wake-up condition after a possible spurious wake-up notification.

Examples

`cnd_wait()` Not Wrapped in Loop

```
#include <stdio.h>
#include <stddef.h>
#include <threads.h>

#define THRESHOLD 100
```

```
static mtx_t lock;
static cnd_t cond;

void func(int input)
{
    if (thrd_success != mtx_lock(&lock)) {
        /* Handle error */
    }
    /* test condition to pause thread */
    if (input > THRESHOLD) {
        if (thrd_success != cnd_wait(&cond, &lock)) {
            /* Handle error */
        }
    }
    /* Proceed if condition to pause does not hold */


    if (thrd_success != mtx_unlock(&lock)) {
        /* Handle error */
    }
}
```

In this example, the thread uses `cnd_wait()` to pause execution when `input` is greater than `THRESHOLD`. The paused thread can resume if another thread uses `cnd_broadcast()`, which notifies all the threads. This notification causes the thread to wake up even if the pause condition is still true.

Correction — Wrap `cnd_wait()` in a `while` Loop

One possible correction is to wrap `cnd_wait()` in a `while` loop. The loop checks the pause condition after the thread receives a possible spurious wake-up notification.

```
#include <stdio.h>
#include <stddef.h>
#include <threads.h>

#define THRESHOLD 100

static mtx_t lock;
static cnd_t cond;

void func(int input)
{
    if (thrd_success != mtx_lock(&lock)) {
        /* Handle error */
    }
    /* test condition to pause thread */
    while (input > THRESHOLD) {
        if (thrd_success != cnd_wait(&cond, &lock)) {
            /* Handle error */
        }
    }
    /* Proceed if condition to pause does not hold */


    if (thrd_success != mtx_unlock(&lock)) {
        /* Handle error */
```

```
    }
}
```

Result Information

Group: Concurrency
Language: C | C++
Default: Off
Command-Line Syntax: SPURIOUS_WAKEUP_NOT_WRAPPED_IN_LOOP
Impact: Low

See Also

`Find defects (-checkers)` | `Function that can spuriously fail not wrapped in loop` | `Returned value of a sensitive function not checked`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Join or detach of a joined or detached thread

Thread that was previously joined or detached is joined or detached again

Description

This defect occurs when:

- You try to join a thread that was previously joined or detached.
- You try to detach a thread that was previously joined or detached.

The **Result Details** pane describes if the thread was previously joined or detached and also shows previous related events.

For instance, a thread joined with `thrd_join` is then detached with `thrd_detach`:

```
thrd_t id;
...
thrd_join(id, NULL);
thrd_detach(id);
```

Note that a thread is considered as joined only if a previous thread joining is successful. For instance, the thread is not considered as joined in the `if` branch here:

```
thrd_t t;
...
if (thrd_success != thrd_join(t, 0)) {
    /* Thread not considered joined */
}
```

The analysis cannot detect cases where a joined thread detaches itself using, for instance, the `thrd_current()` function.

Risk

The C11 standard (clauses 7.26.5.3 and 7.26.5.6) states that a thread shall not be joined or detached once it was previously joined or detached. Violating these clauses of the standard results in undefined behavior.

Fix

Avoid joining a thread that was already joined or detached previously. Avoid detaching a thread that was already joined or detached.

Examples

Joining a Thread Followed by Detaching the Thread

```
#include <stddef.h>
#include <threads.h>
#include <stdlib.h>
```

```
extern int thread_func(void *arg);

int main (void)
{
  thrd_t t;

  if (thrd_success != thrd_create (&t, thread_func, NULL)) {
    /* Handle error */
    return 0;
  }

  if (thrd_success != thrd_join (t, 0)) {
    /* Handle error */
    return 0;
  }

  if (thrd_success != thrd_detach (t)) {
    /* Handle error */
    return 0;
  }

  return 0;
}
```

In this example, the use of `thrd_detach` on a thread that was previously joined with `thrd_join` leads to undefined behavior.

To avoid compilation errors when running Bug Finder on this example, specify the C11 standard with the option `C standard version (-c-version)`.

Correction – Avoid Detaching a Joined Thread

Remove the prior `thrd_join` or the subsequent `thrd_detach` statement. In this corrected version, the `thrd_detach` statement is removed.

```
#include <stddef.h>
#include <threads.h>
#include <stdlib.h>


extern int thread_func(void *arg);

int main (void)
{
  thrd_t t;

  if (thrd_success != thrd_create (&t, thread_func, NULL)) {
    /* Handle error */
    return 0;
  }

  if (thrd_success != thrd_join (t, 0)) {
    /* Handle error */
    return 0;
  }
```

```
  return 0;
}
```

Joining Thread Created in Detached State

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);


int main() {
    pthread_t id;
    pthread_attr_t attr;

    if(thread_success != pthread_attr_init(&attr)) {
        return 0;
    }

    if(thread_success != pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED)) {
        return 0;
    }

    if(thread_success != pthread_create(&id, &attr, thread_func, NULL)) {
            return 0;
    }

    if(thread_success != pthread_join(id, NULL)) {
            return 0;
    }

    return 0;
}
```

In this example, the thread attribute is assigned the state `PTHREAD_CREATE_DETACHED`. A thread created using this attribute is then joined.

Correction – Create Threads as Joinable

One possible correction is to create a thread with thread attribute assigned to the state `PTHREAD_CREATE_JOINABLE` and then join the thread.

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);


int main() {
    pthread_t id;
    pthread_attr_t attr;

    if(thread_success != pthread_attr_init(&attr)) {
        return 0;
    }
```

```
    if(thread_success != pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE)) {
        return 0;
    }

    if(thread_success != pthread_create(&id, &attr, thread_func, NULL)) {
            return 0;
    }

    if(thread_success != pthread_join(id, NULL)) {
            return 0;
    }

    return 0;
}
```

Check Information

Group: Concurrency
Language: C
Default: Off
Command-Line Syntax: DOUBLE_JOIN_OR_DETACH
Impact: Medium

See Also

Missing or double initialization of thread attribute | Use of undefined thread ID

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019b

Missing lock

Unlock function without lock function

Description

This defect occurs when a task calls an unlock function before calling the corresponding lock function.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait till `my_task` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Risk

A call to an unlock function without a corresponding lock function can indicate a coding error. For instance, perhaps the unlock function does not correspond to the lock function that begins the critical section.

Fix

The fix depends on the root cause of the defect. For instance, if the defect occurs because of a mismatch between lock and unlock function, check the lock-unlock function pair in your Polyspace analysis configuration and fix the mismatch.

See examples of fixes below. To avoid the issue, you can follow the practice of calling the lock and unlock functions in the same module at the same level of abstraction. For instance, in this example, `func` calls the lock and unlock function at the same level but `func2` does not.

```
void func() {
  my_lock();
  {
    ...
  }
  my_unlock();
}

void func2() {
  {
   my_lock();
   ...
  }
  my_unlock();
}
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Missing lock

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset(void)
{
  begin_critical_section();
  global_var = 0;
  end_critical_section();
}

void my_task(void)
{
  global_var += 1;
  end_critical_section();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Specification	
`Configure multitasking manually`	☑	
`Tasks (-entry-points)`	`my_task`, `reset`	
`Critical section details (-critical-section-begin -critical-section-end)`	**Starting routine**	**Ending routine**
	`begin_critical_section`	`end_critical_section`

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points my_task,reset
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

The example has two entry points, `my_task` and `reset`. `my_task` calls `end_critical_section` before calling `begin_critical_section`.

Correction — Provide Lock

One possible correction is to call the lock function `begin_critical_section` before the instructions in the critical section.

```
void begin_critical_section(void);
void end_critical_section(void);
```

```
int global_var;

void reset(void)
{
  begin_critical_section();
  global_var = 0;
  end_critical_section();
}

void my_task(void)
{
    begin_critical_section();
    global_var += 1;
    end_critical_section();
}
```

Lock in Condition

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset() {
    begin_critical_section();
    global_var=0;
    end_critical_section();
}

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      if(index%10==0) {
        begin_critical_section();
        global_var ++;
      }
      end_critical_section();
      index++;
    }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Specification	
Configure multitasking manually	☑	
Tasks (-entry-points)	my_task, reset	
Critical section details (-critical-section-begin -critical-section-end)	**Starting routine**	**Ending routine**

Option	Specification	
	`begin_critical_section`	`end_critical_section`

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points my_task,reset
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

The example has two entry points, `my_task` and `reset`.

In the `while` loop, `my_task` leaves a critical section through the call `end_critical_section();`. In an iteration of the `while` loop:

- If `my_task` enters the `if` condition branch, the critical section begins through a call to `begin_critical_section`.
- If `my_task` does not enter the `if` condition branch and leaves the `while` loop, the critical section does not begin. Therefore, a **Missing lock** defect occurs.
- If `my_task` does not enter the `if` condition branch and continues to the next iteration of the `while` loop, the unlock function `end_critical_section` is called again. A **Double unlock** defect occurs.

Because `numCycles` is a `volatile` variable, it can take any value. Any of the cases above are possible. Therefore, a **Missing lock** defect and a **Double unlock** defect appear on the call `end_critical_section`.

Check Information

Group: Concurrency
Language: C | C++
Default: On
Command-Line Syntax: `BAD_UNLOCK`
Impact: Medium
CWE ID: 832

See Also

`Configure multitasking manually` | `Critical section details (-critical-section-begin -critical-section-end)` | `Data race` | `Data race including atomic operations` | `Data race through standard library function call` | `Deadlock` | `Destruction of locked mutex` | `Double lock` | `Double unlock` | `Find defects (-checkers)` | `Missing unlock` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics

“Configuring Polyspace Multitasking Analysis Manually”
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”

Introduced in R2014b

Missing or double initialization of thread attribute

Duplicated initialization of thread attributes or noninitialized thread attribute used in functions that expect initialized attributes

Description

This defect occurs during one of these situations:

- You initialize a thread attribute twice with a function such as `pthread_attr_init` without an intermediate call to a function such as `pthread_attr_destroy`.

  The function `pthread_attr_destroy` destroys a thread attribute object and enables the system to reclaim resources associated with the object.
- You use a noninitialized thread attribute in a function such as `pthread_create`, which expects an initialized attribute. A thread attribute might be noninitialized because it was never initialized previously or destroyed with the `pthread_attr_destroy` function.

  Noninitialized thread attributes are detected for all functions in the POSIX standard.

The **Result Details** pane describes whether the attribute is doubly initialized or noninitialized and also shows previous related events.

Note that a thread attribute is considered initialized only if the call to `pthread_attr_init` is successful. For instance, the thread attribute is not initialized in the `if` branch here:

```
pthread_attr_t attr;
int thread_success;

thread_success = pthread_attr_init(&attr);
if(thread_success != 0) {
   /* Thread attribute considered noninitialized */
}
```

The issue is also flagged if you do not check the return value from a call to `pthread_attr_init`.

Risk

Initializing a thread attribute without destroying the previously initialized attribute or using noninitialized thread attributes leads to undefined behavior.

Fix

Before using a thread attribute, initialize the attribute by using the `pthread_attr_init` function.

```
pthread_attr_t attr;
int thread_success;

/* Initialize attribute */
thread_success = pthread_attr_init(&attr);
if(thread_success != 0) {
   /* Handle initialization error */
}
...
```

```
/* Use attribute */
thread_sucess = pthread_create(&thr, &attr, &thread_start, NULL);
```

After initialization, destroy a thread attribute by using `pthread_attr_destroy` before initializing again:

```
pthread_attr_t attr;
int thread_success;

/* Destroy attribute */
thread_success = pthread_attr_destroy(&attr);
if(thread_success != 0) {
   /* Handle destruction error */
}
...
/* Reinitialize attribute */
thread_success = pthread_attr_init(&attr);
```

Examples

Use of Noninitialized Thread Attribute

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);


int main() {
    pthread_t id;
    pthread_attr_t attr;

    if(thread_success == pthread_create(&id, &attr, thread_func, NULL)) {
    }

    return 0;
}
```

In this example, the attribute `attr` is not initialized before its use in the `pthread_create` call.

Correction – Initialize Thread Attribute Before Use

Before using a thread attribute in the `pthread_create` function, initialize the attribute with the `pthread_attr_init` function.

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);

int main() {
    pthread_t id;
    pthread_attr_t attr;

    if(thread_success != pthread_attr_init(&attr)) {
        return 0;
```

```
    }

    if(thread_success == pthread_create(&id, &attr, thread_func, NULL)) {
    }

    return 0;
}
```

Return Value from Thread Attribute Initialization Not Checked

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);

int main() {
    pthread_t id;
    pthread_attr_t attr;

    pthread_attr_init(&attr);

    if(thread_success == pthread_create(&id, &attr, thread_func, NULL)) {
    }

    return 0;
}
```

In this example, the return value of `pthread_attr_init` is not checked. If the thread attribute initialization fails, the error does not get handled. A possibly undefined thread attribute is later used in the `pthread_create` function.

Correction – Handle Errors from Thread Attribute Initialization

One possible correction is to use the thread attribute only if the return value from `pthread_attr_init` indicates successful initialization.

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);

int main() {
    pthread_t id;
    pthread_attr_t attr;

    if(thread_success != pthread_attr_init(&attr)) {
        return 0;
    }

    if(thread_success == pthread_create(&id, &attr, thread_func, NULL)) {
    }

    return 0;
}
```

Check Information

Group: Concurrency
Language: C
Default: Off
Command-Line Syntax: `BAD_THREAD_ATTRIBUTE`
Impact: Medium

See Also

`Join or detach of a joined or detached thread|Use of undefined thread ID`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019b

Missing unlock

Lock function without unlock function

Description

This defect occurs when:

- A task calls a lock function.
- The task ends without a call to an unlock function.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task, `my_task`, calls a lock function, `my_lock`, other tasks calling `my_lock` must wait until `my_task` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, before analysis, you must specify the multitasking options. On the **Configuration** pane, select **Multitasking**.

Risk

An unlock function ends a critical section so that other waiting tasks can enter the critical section. A missing unlock function can result in tasks blocked for an unnecessary length of time.

Fix

Identify the critical section of code, that is, the section that you want to be executed as an atomic block. At the end of this section, call the unlock function that corresponds to the lock function used at the beginning of the section.

There can be other reasons and corresponding fixes for the defect. Perhaps you called the incorrect unlock function. Check the lock-unlock function pair in your Polyspace analysis configuration and fix the mismatch.

See examples of fixes below. To avoid the issue, you can follow the practice of calling the lock and unlock functions in the same module at the same level of abstraction. For instance, in this example, `func` calls the lock and unlock function at the same level but `func2` does not.

```
void func() {
  my_lock();
  {
    ...
  }
  my_unlock();
}

void func2() {
  {
   my_lock();
   ...
  }
  my_unlock();
}
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Missing Unlock

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset()
{
    begin_critical_section();
    global_var = 0;
    end_critical_section();
}

void my_task(void)
{
    begin_critical_section();
    global_var += 1;
}
```

In this example, to emulate multitasking behavior, specify the following options:

Option	Value	
Configure multitasking manually	☑	
Tasks (-entry-points)	my_task, reset	
Critical section details (-critical-section-begin -critical-section-end)	**Starting routine**	**Ending routine**
	begin_critical_section	end_critical_section

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points my_task,reset
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

The example has two entry points, `my_task` and `reset`. `my_task` enters a critical section through the call `begin_critical_section();`. `my_task` ends without calling `end_critical_section`.

Correction — Provide Unlock

One possible correction is to call the unlock function `end_critical_section` after the instructions in the critical section.

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset(void)
{
    begin_critical_section();
    global_var = 0;
    end_critical_section();
}

void my_task(void)
{
    begin_critical_section();
    global_var += 1;
    end_critical_section();
}
```

Unlock in Condition

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset() {
    begin_critical_section();
    global_var=0;
    end_critical_section();
}

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      begin_critical_section();
      global_var ++;
      if(index%10==0) {
        global_var = 0;
        end_critical_section();
      }
      index++;
    }
}
```

In this example, to emulate multitasking behavior, specify the following options.

Option	Specification
`Configure multitasking manually`	☑

Option	Specification	
Tasks (-entry-points)	my_task, reset	
Critical section details (-critical-section-begin -critical-section-end)	**Starting routine**	**Ending routine**
	begin_critical_section	end_critical_section

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points my_task,reset
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

The example has two entry points, `my_task` and `reset`.

In the `while` loop, `my_task` enters a critical section through the call `begin_critical_section();`. In an iteration of the `while` loop:

- If `my_task` enters the `if` condition branch, the critical section ends through a call to `end_critical_section`.
- If `my_task` does not enter the `if` condition branch and leaves the `while` loop, the critical section does not end. Therefore, a **Missing unlock** defect occurs.
- If `my_task` does not enter the `if` condition branch and continues to the next iteration of the `while` loop, the lock function `begin_critical_section` is called again. A **Double lock** defect occurs.

Because `numCycles` is a `volatile` variable, it can take any value. Any of the cases above is possible. Therefore, a **Missing unlock** defect and a **Double lock** defect appear on the call `begin_critical_section`.

Correction — Place Unlock Outside Condition

One possible correction is to call the unlock function `end_critical_section` outside the `if` condition.

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset() {
    begin_critical_section();
    global_var=0;
    end_critical_section();
}

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
```

```
      begin_critical_section();
      global_var ++;
      if(index%10==0) {
        global_var=0;
      }
      end_critical_section();
      index++;
    }
}
```

Correction — Place Unlock in Every Conditional Branch

Another possible correction is to call the unlock function `end_critical_section` in every branches of the `if` condition.

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset() {
    begin_critical_section();
    global_var=0;
    end_critical_section();
}

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      begin_critical_section();
      global_var ++;
      if(index%10==0) {
        global_var=0;
        end_critical_section();
      }
      else
        end_critical_section();
      index++;
    }
}
```

Check Information

Group: Concurrency
Language: C | C++
Default: On
Command-Line Syntax: `BAD_LOCK`
Impact: High
CWE ID: 667

See Also

`Configure multitasking manually` | `Critical section details (-critical-section-begin -critical-section-end)` | `Data race` | `Data race including atomic operations` | `Data race through standard library function call` | `Deadlock` | `Destruction of locked mutex` | `Double lock` | `Double unlock` | `Find defects (-checkers)` | `Missing lock` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics

“Configuring Polyspace Multitasking Analysis Manually”
“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”

Introduced in R2014b

Multiple mutexes used with same condition variable

Threads using different mutexes when concurrently waiting on the same condition variable is undefined behavior

Description

This defect occurs when multiple threads use more than one mutex to concurrently wait on the same condition variable. A thread waits on a condition variable by calling the functions `pthread_cond_timedwait` or `pthread_cond_wait`. These functions take a condition variable and a locked mutex as arguments, and the condition variable is bound to that mutex when the thread waits on the condition variable.

The checkers flags the use of `pthread_cond_timedwait` or `pthread_cond_wait` in one of the threads. See the **Event** column in the **Results Details** pane to view the threads waiting on the same condition variable and using a different mutex.

Risk

When a thread waits on a condition variable using a mutex, the condition variable is bound to that mutex. Any other thread using a different mutex to wait on the same condition variable is undefined behavior according to the POSIX standard.

Fix

Use the same mutex argument for `pthread_cond_timedwait` or `pthread_cond_wait` when threads are concurrently waiting on the same condition variable, or use separate condition variables for each mutex.

Examples

Concurrent Waiting on Condition Variable with Multiple Mutexes

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define Thrd_return_t                   void *
#define __USE_XOPEN2K8



#define COUNT_LIMIT 5

static void fatal_error(void)
{
    exit(1);
}


pthread_mutex_t mutex1;
pthread_mutex_t mutex2;
pthread_mutex_t mutex3;
pthread_cond_t cv;

int count1 = 0, count2 = 0, count3 = 0;
#define DELAY 8
```

```
Thrd_return_t waiter1(void* arg)
{
    int ret;
    while (count1 < COUNT_LIMIT) {
        if ((ret = pthread_mutex_lock(&mutex1)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                  pthread_cond_wait(&cv, &mutex1)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count1 = %d\n", ++count1);
        if ((ret = pthread_mutex_unlock(&mutex1)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t waiter2(void* arg)
{
    int ret;
    while (count2 < COUNT_LIMIT) {
        if ((ret = pthread_mutex_lock(&mutex2)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                  pthread_cond_wait(&cv, &mutex2)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count2 = %d\n", ++count2);
        if ((ret = pthread_mutex_unlock(&mutex2)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t signaler(void* arg)
{
    int ret;
    while ((count1 < COUNT_LIMIT) || (count2 < COUNT_LIMIT)) {
        sleep(1);
        printf("signaling\n");
        if ((ret = pthread_cond_broadcast(&cv)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t waiter3(void* arg)
{
    int ret;
    while (count3 % COUNT_LIMIT != 0) {
        if ((ret = pthread_mutex_lock(&mutex3)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                  pthread_cond_wait(&cv, &mutex3)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count3 = %d\n", ++count3);
        if ((ret = pthread_mutex_unlock(&mutex3)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
```

```
}

int main(void)
{
    int ret;
    pthread_t thread1, thread2, thread3;

    pthread_mutexattr_t attr;

    if ((ret = pthread_mutexattr_init(&attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
        /* Handle error */
        fatal_error();
    }

    if ((ret = pthread_mutex_init(&mutex1, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutex_init(&mutex2, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutex_init(&mutex3, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_cond_init(&cv, NULL)) != 0) {
        /* handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread1, NULL, &waiter1, NULL))) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread2, NULL, &waiter2, NULL))) {
        /* handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread3, NULL, &signaler, NULL))) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread1, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread2, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread3, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }

    while (1) { ; }

    return 0;
}
```

In this example, a different mutex is used to protect each `count` variable. Since all three `waiter` functions wait on the same condition variable `cv` with different mutexes, the call to `pthread_cond_wait` will succeed for one of the threads and the call will be undefined for the other two.

The checker raises a defect for function `waiter3` even though the function is not invoked directly or indirectly by a thread, entry-point, or interrupt. The analysis considers function `waiter3` called by the main program through its function address or an unidentified thread whose creation is the missing source code.

Correction — Use the Same Mutex for All Threads Waiting on Same Condition Variable

One possible correction is to pass the same mutex argument to all the call to `pthread_cond_wait` that are used to wait on the same condition variable.

```
 #include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define Thrd_return_t                   void *
#define __USE_XOPEN2K8



#define COUNT_LIMIT 5

static void fatal_error(void)
{
    exit(1);
}


pthread_mutex_t mutex;

pthread_cond_t cv;

int count1 = 0, count2 = 0, count3 = 0;
#define DELAY 8

Thrd_return_t waiter1(void* arg)
{
    int ret;
    while (count1 < COUNT_LIMIT) {
        if ((ret = pthread_mutex_lock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                  pthread_cond_wait(&cv, &mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count1 = %d\n", ++count1);
        if ((ret = pthread_mutex_unlock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t waiter2(void* arg)
{
    int ret;
    while (count2 < COUNT_LIMIT) {
        if ((ret = pthread_mutex_lock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                  pthread_cond_wait(&cv, &mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count2 = %d\n", ++count2);
        if ((ret = pthread_mutex_unlock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t signaler(void* arg)
{
    int ret;
    while ((count1 < COUNT_LIMIT) || (count2 < COUNT_LIMIT)) {
        sleep(1);
```

```
        printf("signaling\n");
        if ((ret = pthread_cond_broadcast(&cv)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t waiter3(void* arg)
{
    int ret;
    while (count3 % COUNT_LIMIT != 0) {
        if ((ret = pthread_mutex_lock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                pthread_cond_wait(&cv, &mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count3 = %d\n", ++count3);
        if ((ret = pthread_mutex_unlock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}
/*
void user_task(void)
{
    (void)waiter3(NULL);
} */

int main(void)
{
    int ret;
    pthread_t thread1, thread2, thread3;

    pthread_mutexattr_t attr;

    if ((ret = pthread_mutexattr_init(&attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
        /* Handle error */
        fatal_error();
    }

    if ((ret = pthread_mutex_init(&mutex, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutex_init(&mutex, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutex_init(&mutex, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_cond_init(&cv, NULL)) != 0) {
        /* handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread1, NULL, &waiter1, NULL))) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread2, NULL, &waiter2, NULL))) {
        /* handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread3, NULL, &signaler, NULL))) {
        /* Handle error */
        fatal_error();
    }
```

```
    if ((ret = pthread_join(thread1, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread2, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread3, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }

    while (1) { ; }

    return 0;
}
```

Result Information

Group: Concurrency
Language: C | C++
Default: Off
Command-Line Syntax: `MULTI_MUTEX_WITH_ONE_COND_VAR`
Impact: Medium

See Also

`Data race` | `Deadlock` | `Find defects (-checkers)` | `Function that can spuriously fail not wrapped in loop` | `Function that can spuriously wake up not wrapped in loop` | `Missing lock` | `Missing unlock` | `Multiple threads waiting on same condition variable`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

External Websites

POS53-C

Introduced in R2020a

Signal call in multithreaded program

Program with multiple threads uses `signal` function

Description

This defect occurs when you use the `signal()` function in a program with multiple threads.

Risk

According to the C11 standard (Section 7.14.1.1), use of the `signal()` function in a multithreaded program is undefined behavior.

Fix

Depending on your intent, use other ways to perform an asynchronous action on a specific thread.

Examples

Use of `signal()` Function to Terminate Loop in Thread

```
#include <signal.h>
#include <stddef.h>
#include <threads.h>

volatile sig_atomic_t flag = 0;

void handler(int signum) {
  flag = 1;
}

/* Runs until user sends SIGUSR1 */
int func(void *data) {
  while (!flag) {
    /* ... */
  }
  return 0;
}

int main(void) {
  signal(SIGINT, handler); /* Undefined behavior */
  thrd_t tid;

  if (thrd_success != thrd_create(&tid, func, NULL)) {
    /* Handle error */
  }
  /* ... */
  return 0;
}
```

In this example, the `signal` function is used to terminate a `while` loop in the thread created with `thrd_create`.

Correction — Use `atomic_bool` Variable to Terminate Loop

One possible correction is to use an `atomic_bool` variable that multiple threads can access. In the corrected example, the child thread evaluates this variable before every loop iteration. After completing the program, you can modify this variable so that the child thread exits the loop.

```
#include <stdatomic.h>
#include <stdbool.h>
#include <stddef.h>
#include <threads.h>

atomic_bool flag = ATOMIC_VAR_INIT(false);

int func(void *data) {
  while (!flag) {
    /* ... */
  }
  return 0;
}

int main(void) {
  thrd_t tid;

  if (thrd_success != thrd_create(&tid, func, NULL)) {
    /* Handle error */
  }
  /* ... */
  /* Set flag when done */
  flag = true;

  return 0;
}
```

Result Information

Group: Concurrency
Language: C | C++
Default: Off
Command-Line Syntax: `SIGNAL_USE_IN_MULTITHREADED_PROGRAM`
Impact: Low

See Also

`Find defects (-checkers)` | `Function called from signal handler not asynchronous-safe` | `MISRA C:2012 Rule 21.5` | `Signal call from within signal handler`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Thread-specific memory leak

Dynamically allocated thread-specific memory not freed before end of thread

Description

This defect occurs when you do not free thread-specific dynamically allocated memory before the end of a thread.

To create thread-specific storage, you generally do these steps:

1. You create a key for thread-specific storage.
2. You create the threads.
3. In each thread, you allocate storage dynamically and then associate the key with this storage.

   After the association, you can read the stored data later using the key.
4. Before the end of the thread, you free the thread-specific memory using the key.

The checker flags execution paths in the thread where the last step is missing.

The checker works on these families of functions:

- `tss_get` and `tss_set` (C11)
- `pthread_getspecific` and `pthread_setspecific` (POSIX)

Risk

The data stored in the memory is available to other processes even after the threads end (memory leak). Besides security vulnerabilities, memory leaks can shrink the amount of available memory and reduce performance.

Fix

Free dynamically allocated memory before the end of a thread.

You can explicitly free dynamically allocated memory with functions such as `free`.

Alternatively, when you create a key, you can associate a destructor function with the key. The destructor function is called with the key value as argument at the end of a thread. In the body of the destructor function, you can free any memory associated with the key. If you use this method, Bug Finder still flags a defect. Ignore this defect with appropriate comments. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Memory Not Freed at End of Thread

```
#include <threads.h>
#include <stdlib.h>

/* Global key to the thread-specific storage */
```

```
tss_t key;
enum { MAX_THREADS = 3 };


int add_data(void) {
  int *data = (int *)malloc(2 * sizeof(int));
  if (data == NULL) {
    return -1;  /* Report error  */
  }
  data[0] = 0;
  data[1] = 1;

  if (thrd_success != tss_set(key, (void *)data)) {
    /* Handle error */
  }
  return 0;
}

void print_data(void) {
  /* Get this thread's global data from key */
  int *data = tss_get(key);

  if (data != NULL) {
    /* Print data */
  }
}

int func(void *dummy) {
  if (add_data() != 0) {
    return -1;  /* Report error */
  }
  print_data();
  return 0;
}

int main(void) {
  thrd_t thread_id[MAX_THREADS];

  /* Create the key before creating the threads */
  if (thrd_success != tss_create(&key, NULL)) {
    /* Handle error */
  }

  /* Create threads that would store specific storage */
  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_create(&thread_id[i], func, NULL)) {
      /* Handle error */
    }
  }

  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_join(thread_id[i], NULL)) {
      /* Handle error */
    }
  }

  tss_delete(key);
```

```
  return 0;
}
```

In this example, the start function of each thread `func` calls two functions:

- `add_data`: This function allocates storage dynamically and associates the storage with a key using the `tss_set` function.
- `print_data`: This function reads the stored data using the `tss_get` function.

At the points where `func` returns, the dynamically allocated storage has not been freed.

Correction — Free Dynamically Allocated Memory Explicitly

One possible correction is to free dynamically allocated memory explicitly before leaving the start function of a thread. See the highlighted change in the corrected version.

In this corrected version, a defect still appears on the `return` statement in the error handling section of `func`. The defect cannot occur in practice because the error handling section is entered only if dynamic memory allocation fails. Ignore this remaining defect with appropriate comments. See "Address Polyspace Results Through Bug Fixes or Justifications".

```
#include <threads.h>
#include <stdlib.h>

/* Global key to the thread-specific storage */
tss_t key;
enum { MAX_THREADS = 3 };


int add_data(void) {
  int *data = (int *)malloc(2 * sizeof(int));
  if (data == NULL) {
    return -1;  /* Report error  */
  }
  data[0] = 0;
  data[1] = 1;

  if (thrd_success != tss_set(key, (void *)data)) {
    /* Handle error */
  }
  return 0;
}

void print_data(void) {
  /* Get this thread's global data from key */
  int *data = tss_get(key);

  if (data != NULL) {
    /* Print data */
  }
}

int func(void *dummy) {
  if (add_data() != 0) {
    return -1;  /* Report error */
  }
  print_data();
```

```
  free(tss_get(key));
  return 0;
}

int main(void) {
  thrd_t thread_id[MAX_THREADS];

  /* Create the key before creating the threads */
  if (thrd_success != tss_create(&key, NULL)) {
    /* Handle error */
  }

  /* Create threads that would store specific storage */
  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_create(&thread_id[i], func, NULL)) {
      /* Handle error */
    }
  }

  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_join(thread_id[i], NULL)) {
      /* Handle error */
    }
  }

  tss_delete(key);
  return 0;
}
```

Result Information

Group: Concurrency
Language: C | C++
Default: Off
Command-Line Syntax: THREAD_MEM_LEAK
Impact: Medium
CWE ID: 401, 404

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Use of signal to kill thread

Uncaught signal kills entire process instead of specific thread

Description

This defect occurs when you use an uncaught signal to kill a thread. For instance, you use the POSIX function `pthread_kill` and send the signal `SIGTERM` to kill a thread.

Risk

Sending a signal kills the entire process instead of just the thread that you intend to kill.

For instance, the `pthread_kill` specifications state that if the disposition of a signal is to terminate, this action affects the entire process.

Fix

Use other mechanisms that are intended to kill specific threads.

For instance, use the POSIX function `pthread_cancel` to terminate a specific thread.

Examples

Use of `pthread_kill` to Terminate Threads

```
#include <signal.h>
#include <pthread.h>

void* func(void *foo) {
  /* Execution of thread */
}

int main(void) {
  int result;
  pthread_t thread;

  if ((result = pthread_create(&thread, NULL, func, 0)) != 0) {
  }
  if ((result = pthread_kill(thread, SIGTERM)) != 0) {
  }

  /* This point is not reached because the process terminates in pthread_kill() */

  return 0;
}
```

In this example, the `pthread_kill` function sends the signal `SIGTERM` to kill a thread. The signal kills the entire process instead of the thread previously created with `pthread_create`.

Correction — Use `pthread_cancel` to Terminate Threads

One possible correction is to use the `pthread_cancel` function. The `pthread_cancel` terminates a thread specified by its first argument at a specific cancellation point or immediately, depending on the thread's cancellation type.

```
#include <signal.h>
#include <pthread.h>

void* func(void *foo) {
  /* Execution of thread */
}

int main(void) {
  int result;
  pthread_t thread;

  if ((result = pthread_create(&thread, NULL, func, 0)) != 0) {
    /* Handle Error */
  }
  if ((result = pthread_cancel(thread)) != 0) {
    /* Handle Error */
  }

  /* Continue executing */

  return 0;
}
```

See also:

- `pthread_cancel` for more information on cancellation types.
- Pthreads for functions that are allowed to be cancellation points.

Result Information

Group: Concurrency
Language: C | C++
Default: Off
Command-Line Syntax: `THREAD_KILLED_WITH_SIGNAL`
Impact: Low

See Also

`Find defects (-checkers)` | `Signal call in multithreaded program`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Use of undefined thread ID

Thread ID from failed thread creation used in subsequent thread functions

Description

This defect occurs when a thread creation function such as `pthread_create` fails but you continue to use the ID from the thread creation.

For instance, `pthread_join` uses an undefined thread ID after the previous thread creation failed. The nonzero return value from `pthread_create` indicates the failed thread creation.

```
pthread_t id;
if(0! = pthread_create(&id, attr, start_func, NULL)) {
    ...
    phread_join(id, NULL);
    ...
}
```

The issue is also flagged if you do not check the return value from a call to `pthread_create`.

Risk

According to the POSIX standard, if thread creation fails, the contents of the thread ID are undefined. The use of an undefined thread ID can lead to unpredictable results.

The issue often indicates a programming error. For instance, it is possible that you tested for nonzero values to determine successful thread creation:

```
if(0 != pthread_create(&id, attr, start_func, NULL))
```

instead of zero:

```
if(0 == pthread_create(&id, attr, start_func, NULL))
```

Fix

If the use of an undefined thread ID comes from a programming error, fix the error. Otherwise, remove the thread functions that are using the undefined ID.

Examples

Threads Joined After Failed Thread Creation

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);


int main() {
    pthread_t id;
    if(thread_success != pthread_create(&id, NULL, thread_func, NULL)) {
```

```
          if(thread_success == pthread_join(id, NULL)) {
          }
    }

    return 0;
}
```

In this example, if `pthread_create` returns a nonzero value, thread creation has failed. The value of `*id` is undefined. The subsequent call to `pthread_join` uses this undefined value.

Correction – Join Threads After Successful Thread Creation

One possible correction is to call `pthread_join` with the thread ID as argument only if `pthread_create` returns zero.

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);


int main() {
    pthread_t id;
    if(thread_success == pthread_create(&id, NULL, thread_func, NULL)) {
          if(thread_success == pthread_join(id, NULL)) {
          }
    }

    return 0;
}
```

Return Value from Thread Creation Not Checked

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);


int main() {
    pthread_t id;
    pthread_create(&id, NULL, thread_func, NULL);
    if(thread_success == pthread_join(id, NULL)) {
    }

    return 0;
}
```

In this example, the return value of `pthread_create` is not checked. If thread creation fails, the error does not get handled. A possibly undefined thread ID is later used in the `pthread_join` function.

Correction – Handle Errors from Thread Creation

One possible correction is to use the ID from thread creation only if the return value from `pthread_create` indicates successful thread creation.

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);


int main() {
    pthread_t id;
    if(thread_success == pthread_create(&id, NULL, thread_func, NULL)) {
         if(thread_success == pthread_join(id, NULL)) {
         }
    }

    return 0;
}
```

Check Information

Group: Concurrency
Language: C
Default: Off
Command-Line Syntax: `UNDEFINED_THREAD_ID`
Impact: Medium

See Also

`Join or detach of a joined or detached thread|Missing or double initialization of thread attribute`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019b

Object Oriented Defects

*this not returned in copy assignment operator

`operator=` method does not return a pointer to the current object

Description

This defect occurs when assignment operators such as `operator=` and `operator+=` do not return a reference to `*this`, where `this` is a pointer to the current object. If the `operator=` method does not return `*this`, it means that `a=b` or `a.operator=(b)` is not returning the assignee `a` following the assignment.

For instance:

- The operator returns its parameter instead of a reference to the current object.

  That is, the operator has a form `MyClass & operator=(const MyClass & rhs) { ... return rhs; }` instead of `MyClass & operator=(const MyClass & rhs) { ... return *this; }`.
- The operator returns by value and not reference.

  That is, the operator has a form `MyClass operator=(const MyClass & rhs) { ... return *this; }` instead of `MyClass & operator=(const MyClass & rhs) { ... return *this; }`.

Risk

Users typically expect object assignments to behave like assignments between built-in types and expect an assignment to return the assignee. For instance, a right-associative chained assignment `a=b=c` requires that `b=c` return the assignee `b` following the assignment. If your assignment operator behaves differently, users of your class can face unexpected consequences.

The unexpected consequences occur when the assignment is part of another statement. For instance:

- If the `operator=` returns its parameter instead of a reference to the current object, the assignment `a=b` returns `b` instead of `a`. If the `operator=` performs a partial assignment of data members, following an assignment `a=b`, the data members of `a` and `b` are different. If you or another user of your class read the data members of the return value and expect the data members of `a`, you might have unexpected results. For an example, see “Return Value of operator= Same as Argument” on page 3-654.
- If the `operator=` method returns `*this` by value and not reference, a copy of `*this` is returned. If you expect to modify the result of the assignment using a statement such as `(a=b).modifyValue()`, you modify a copy of `a` instead of `a` itself.

Fix

Return `*this` from your assignment operators.

Examples

Return Value of operator= Same as Argument

```
class MyClass {
    public:
```

```
        MyClass(bool b, int i): m_b(b), m_i(i) {}
        const MyClass& operator=(const MyClass& obj) {
            if (&obj!=this) {
                /* Note: Only m_i is copied. m_b retains its original value. */
                m_i = obj.m_i;
            }
            return obj;
        }
        bool isOk() const { return m_b;}
        int getI() const { return m_i;}
    private:
        bool m_b;
        int m_i;
};

void main() {
        MyClass r0(true, 0), r1(false, 1);
        /* Object calling isOk is r0 and the if block executes. */
        if ( (r1 = r0).isOk()) {
            /* Do something */
        }
}
```

In this example, the operator `operator=` returns its current argument instead of a reference to `*this`.

Therefore, in `main`, the assignment `r1 = r0` returns `r0` and not `r1`. Because the `operator=` does not copy the data member `m_b`, the value of `r0.m_b` and `r1.m_b` are different. The following unexpected behavior occurs.

What You Might Expect	What Actually Happens
• The statement `(r1 = r0).isOk()` returns `r1.m_b` which has value `false` • The `if` block does not execute.	• The statement `(r1 = r0).isOk()` returns `r0.m_b` which has value `true` • The `if` block executes.

Correction — Return `*this`

One possible correction is to return `*this` from `operator=`.

```
class MyClass {
    public:
        MyClass(bool b, int i): m_b(b), m_i(i) {}
        const MyClass& operator=(const MyClass& obj) {
            if (&obj!=this) {
                /* Note: Only m_i is copied. m_b retains its original value. */
                m_i = obj.m_i;
            }
            return *this;
        }
        bool isOk() const { return m_b;}
        int getI() const { return m_i;}
    private:
        bool m_b;
        int m_i;
};
```

```
void main() {
        MyClass r0(true, 0), r1(false, 1);
        /* Object calling isOk is r0 and the if block executes. */
        if ( (r1 = r0).isOk()) {
            /* Do something */
        }
}
```

Result Information

Group: Object oriented
Language: C++
Default: Off
Command-Line Syntax: `RETURN_NOT_REF_TO_THIS`
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Base class assignment operator not called

Copy assignment operator does not call copy assignment operators of base subobjects

Description

This defect occurs when a derived class copy assignment operator does not call the copy assignment operator of its base class.

Risk

If this defect occurs, unless you are initializing the base class data members explicitly in the derived class assignment operator, the operator initializes the members implicitly by using the default constructor of the base class. Therefore, it is possible that the base class data members do not get assigned the right values.

If users of your class expect your assignment operator to perform a complete assignment between two objects, they can face unintended consequences.

Fix

Call the base class copy assignment operator from the derived class copy assignment operator.

Even if the base class data members are not `private`, and you explicitly initialize the base class data members in the derived class copy assignment operator, replace this explicit initialization with a call to the base class copy assignment operator. Otherwise, determine why you retain the explicit initialization.

Examples

Base Class Copy Assignment Operator Not Called

```
class Base0 {
public:
    Base0();
    virtual ~Base0();
    Base0& operator=(const Base0&);
private:
    int _i;
};

class Base1 {
public:
    Base1();
    virtual ~Base1();
    Base1& operator=(const Base1&);
private:
    int _i;
};

class Derived: public Base0, Base1 {
public:
    Derived();
```

```
    ~Derived();
    Derived& operator=(const Derived& d) {
        if (&d == this) return *this;
        Base0::operator=(d);
        _j = d._j;
        return *this;
    }
private:
    int _j;
};
```

In this example, the class `Derived` is derived from two classes `Base0` and `Base1`. In the copy assignment operator of `Derived`, only the copy assignment operator of `Base0` is called. The copy assignment operator of `Base1` is not called.

The defect appears on the copy assignment operator of the derived class. Following are some tips for navigating in the source code:

- To find the derived class definition, right-click the derived class name and select **Go To Definition**.
- To find the base class definition, first navigate to the derived class definition. In the derived class definition, right-click the base class name and select **Go To Definition**.
- To find the definition of the base class copy assignment operator, first navigate to the base class definition. In the base class definition, right-click the operator name and select **Go To Definition**.

Correction — Call Base Class Copy Assignment Operator

If you want your copy assignment operator to perform a complete assignment, one possible correction is to call the copy assignment operator of class `Base1`.

```
class Base0 {
public:
    Base0();
    virtual ~Base0();
    Base0& operator=(const Base0&);
private:
    int _i;
};

class Base1 {
public:
    Base1();
    virtual ~Base1();
    Base1& operator=(const Base1&);
private:
    int _i;
};

class Derived: public Base0, Base1 {
public:
    Derived();
    ~Derived();
    Derived& operator=(const Derived& d) {
        if (&d == this) return *this;
        Base0::operator=(d);
        Base1::operator=(d);
        _j = d._j;
```

```
        return *this;
    }
private:
    int _j;
};
```

Result Information

Group: Object oriented
Language: C++
Default: On
Command-Line Syntax: `MISSING_BASE_ASSIGN_OP_CALL`
Impact: High

See Also

`Copy constructor not called in initialization list|Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Base class destructor not virtual

Class cannot behave polymorphically for deletion of derived class objects

Description

This defect occurs when a class has `virtual` functions but not a `virtual` destructor.

Risk

The presence of `virtual` functions indicates that the class is intended for use as a base class. However, if the class does not have a `virtual` destructor, it cannot behave polymorphically for deletion of derived class objects.

If a pointer to this class refers to a derived class object, and you use the pointer to delete the object, only the base class destructor is called. Additional resources allocated in the derived class are not released and can cause a resource leak.

Fix

One possible fix is to always use a `virtual` destructor in a class that contains `virtual` functions.

Examples

Base Class Destructor Not Virtual

```
class Base {
        public:
                Base(): _b(0) {};
                virtual void update() {_b += 1;};
        private:
                int _b;
};

class Derived: public Base {
        public:
                Derived(): _d(0) {};
                ~Derived() {_d = 0;};
                virtual void update() {_d += 1;};
        private:
                int _d;
};
```

In this example, the class `Base` does not have a `virtual` destructor. Therefore, if a `Base*` pointer points to a `Derived` object that is allocated memory dynamically, and the `delete` operation is performed on that `Base*` pointer, the `Base` destructor is called. The memory allocated for the additional member `_d` is not released.

The defect appears on the base class definition. Following are some tips for navigating in the source code:

- To find classes derived from the base class, right-click the base class name and select **Search For All References**. Browse through each search result to find derived class definitions.

- To find if you are using a pointer or reference to a base class to point to a derived class object, right-click the base class name and select **Search For All References**. Browse through search results that start with `Base*` or `Base&` to locate pointers or references to the base class. You can then see if you are using a pointer or reference to point to a derived class object.

Correction — Make Base Class Destructor Virtual

One possible correction is to declare a `virtual` destructor for the class `Base`.

```
class Base {
        public:
                Base(): _b(0) {};
                virtual ~Base() {_b = 0;};
                virtual void update() {_b += 1;};
        private:
                int _b;
};

class Derived: public Base {
        public:
                Derived(): _d(0) {};
                ~Derived() {_d = 0;};
                virtual void update() {_d += 1;};
        private:
                int _d;
};
```

Result Information

Group: Object oriented
Language: C++
Default: On
Command-Line Syntax: `DTOR_NOT_VIRTUAL`
Impact: Medium

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

External Websites

CERT C++ OOP52-CPP

Introduced in R2015b

Bytewise operations on nontrivial class object

Value representations may be improperly initialized or compared

Description

This defect occurs when you use C Standard library functions to perform bytewise operation on non-trivial or non-standard layout class type objects. For definitions of trivial and standard layout classes, see the C++ Standard, [class], paragraphs 6 and 7 respectively.

The checker raises a defect when you initialize or copy non-trivial class type objects using these functions:

- `std::memset`
- `std::memcpy`
- `std::strcpy`
- `std::memmove`

Or when you compare non-standard layout class type objects using these functions:

- `std::memcmp`
- `std::strcmp`

The checker raises no defect if the bytewise operation is performed through an alias. For example no defect is raised in the bytewise comparison and copy operations in this code. The bytewise operations use `dptr` and `sptr`, the aliases of non-trivial or non-standard layout class objects `d` and `s`.

```
void func(NonTrivialNonStdLayout *d, const NonTrivialNonStdLayout *s)
{
   void* dptr = (void*)d;
   const void* sptr = (void*)s;
   // ...
   // ...
   // ...
   if (!std::memcmp(dptr, sptr, sizeof(NonTrivialNonStdLayout))) {
     (void)std::memcpy(dptr, sptr, sizeof(NonTrivialNonStdLayout));
      // ...
   }
}
```

Risk

Performing bytewise comparison operations by using C Standard library functions on non-trivial or non-standard layout class type object might result in unexpected values due to implementation details. The object representation depends on the implementation details, such as the order of private and public members, or the use of virtual function pointer tables to represent the object.

Performing bytewise setting operations by using C Standard library functions on non-trivial or non-standard layout class type object can change the implementation details. The operation might result in abnormal program behavior or a code execution vulnerability. For instance, if the address of a member function is overwritten, the call to this function invokes an unexpected function.

Fix

To perform bytewise operations non-trivial or non-standard layout class type object, use these C++ special member functions instead of C Standard library functions.

C Standard Library Functions	C++ Member Functions
`std::memset`	Class constructor

`std::memcpy`
`std::strcpy`
`std::memmove`
	Class copy constructor
	Class move constructor
	Copy assignment operator
	Move assignment operator

`std::memcmp`
`std::strcmp`
	`operator<()`
	`operator>()`
	`operator==()`
	`operator!=()`

Examples

Using `memset` with non-trivial class object

```
#include <cstring>
#include <iostream>
#include <utility>

class nonTrivialClass
{
    int scalingFactor;
    int otherData;
public:
    nonTrivialClass() : scalingFactor(1) {}
    void set_other_data(int i);
    int f(int i)
    {
        return i / scalingFactor;
    }
    // ...
};

void func()
{
    nonTrivialClass c;
    // ... Code that mutates c ...
    std::memset(&c, 0, sizeof(nonTrivialClass));
    std::cout << c.f(100) << std::endl;
}
```

In this example, `func()` uses `std::memset` to reinitialize non-trivial class object `c` after it is first initialized with its default constructor. This bytewise operation might not properly initialize the value representation of `c`.

Correction — Define Function Template That Uses `std::swap`

One possible correction is to define a function template `clear()` that uses `std::swap` to perform a swap operation. The call to `clear()`properly reinitializes object `c` by swapping the contents of `c` and default initialized object `empty`.

```
 #include <cstring>
#include <iostream>
#include <utility>

class nonTrivialClass
{
    int scalingFactor;
    int otherData;
public:
    nonTrivialClass() : scalingFactor(1) {}
    void set_other_data(int i);
    int f(int i)
    {
        return i / scalingFactor;
    }
    // ...
};

template <typename T>
T& clear(T& o)
{
    using std::swap;
    T empty;
    swap(o, empty);
    return o;
}

void func()
{
    nonTrivialClass c;
    // ... Code that mutates c ...

    clear(c);
    std::cout << c.f(100) << std::endl;
}
```

Result Information

Group: Object Oriented
Language: C++
Default: Off
Command-Line Syntax: `MEMOP_ON_NONTRIVIAL_OBJ`
Impact: Medium

See Also

`Copy of overlapping memory` | `Find defects (-checkers)` | `Memory comparison of strings` | `Memory comparison of padding data`

Topics

“Interpret Polyspace Bug Finder Results”

“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019b

Conversion or deletion of incomplete class pointer

You delete or cast to a pointer to an incomplete class

Description

This defect occurs when you delete or cast to a pointer to an incomplete class. An incomplete class is one whose definition is not visible at the point where the class is used.

For instance, the definition of class `Body` is not visible when the `delete` operator is called on a pointer to `Body`:

```
class Handle {
  class Body *impl;
public:
  ~Handle() { delete impl; }
  // ...
};
```

Risk

When you delete a pointer to an incomplete class, it is not possible to call any nontrivial destructor that the class might have. If the destructor performs cleanup activities such as memory deallocation, these activities do not happen.

A similar problem happens, for instance, when you downcast to a pointer to an incomplete class (downcasting is casting from a pointer to a base class to a pointer to a derived class). At the point of downcasting, the relationship between the base and derived class is not known. In particular, if the derived class inherits from multiple classes, at the point of downcasting, this information is not available. The downcasting cannot make the necessary adjustments for multiple inheritance and the resulting pointer cannot be dereferenced.

A similar statement can be made for upcasting (casting from a pointer to derived class to a pointer to a base class).

Fix

When you delete or downcast to a pointer to a class, make sure that the class definition is visible.

Alternatively, you can perform one of these actions:

- Instead of a regular pointer, use the `std::shared_ptr` type to point to the incomplete class.
- When downcasting, make sure that the result is valid. Write error-handling code for invalid results.

Examples

Deletion of Pointer to Incomplete Class

```
class Handle {
  class Body *impl;
public:
```

```
  ~Handle() { delete impl; }
  // ...
};
```

In this example, the definition of class `Body` is not visible when the pointer to `Body` is deleted.

Correction — Define Class Before Deletion

One possible correction is to make sure that the class definition is visible when a pointer to the class is deleted.

```
class Handle {
  class Body *impl;
public:
  ~Handle();
  // ...
};

// Elsewhere
class Body { /* ... */ };

Handle::~Handle() {
  delete impl;
}
```

Correction — Use `std::shared_ptr`

Another possible correction is to use the `std::shared_ptr` type instead of a regular pointer.

```
#include <memory>

class Handle {
  std::shared_ptr<class Body> impl;
  public:
    Handle();
    ~Handle() {}
    // ...
};
```

Downcasting to Pointer to Incomplete Class

`File1.h`:

```
class Base {
protected:
  double var;
public:
  Base() : var(1.0) {}
  virtual void do_something();
  virtual ~Base();
};
```

`File2.h`:

```
void funcprint(class Derived *);
class Base *get_derived();
```

`File1.cpp`:

```
#include "File1.h"
#include "File2.h"

void getandprint() {
  Base *v = get_derived();
  funcprint(reinterpret_cast<class Derived *>(v));
}
```

`File2.cpp`:

```
#include "File2.h"
#include "File1.h"
#include <iostream>

class Base2 {
protected:
  short var2;
public:
  Base2() : var2(12) {}
};

class Derived : public Base2, public Base {
  float var_derived;
public:
    Derived() : Base2(), Base(), var_derived(1.2f) {}
    void do_something()
    {
        std::cout << "var_derived: "
                  << var_derived << ", var : " << var
                  << ", var2: " << var2 << std::endl;
    }
 };

void funcprint(Derived *d) {
  d->do_something();
}

Base *get_derived() {
  return new Derived;
}
```

In this example, the definition of class `Derived` is not visible in `File1.cpp` when a `Base*` pointer to downcast to a `Derived*` pointer.

In `File2.cpp`, class `Derived` derives from two classes, `Base` and `Base2`. This information about multiple inheritance is not available at the point of downcasting in `File1.cpp`. The result of downcasting is passed to the function `funcprint` and dereferenced in the body of `funcprint`. Because the downcasting was done with incomplete information, the dereference can be invalid.

Correction — Define Class Before Downcasting

One possible correction is to define the class `Derived` before downcasting a `Base*` pointer to a `Derived*` pointer.

In this corrected example, the downcasting is done in `File2.cpp` in the body of `funcprint` at a point where the definition of class `Derived` is visible. The downcasting is not done in `File1.cpp` where the definition of `Derived` is not visible. The changes from the previous incorrect example are highlighted.

File1.h:

```
class Base {
protected:
  double var;
public:
  Base() : var(1.0) {}
  virtual void do_something();
  virtual ~Base();
};
```

File2.h:

```
void funcprint(class Base *);
class Base *get_derived();
```

File1.cpp:

```
#include "File1.h"
#include "File2.h"

void getandprint() {
  Base *v = get_derived();
  funcprint(v);
}
```

File2.cpp:

```
#include "File2_corr.h"
#include "File1_corr.h"
#include <iostream>

class Base2 {
protected:
  short var2;
public:
  Base2() : var2(12) {}
};

class Derived : public Base2, public Base {
  float var_derived;

public:
    Derived() : Base2(), Base(), var_derived(1.2f) {}
    void do_something()
    {
        std::cout << "var_derived: "
                  << var_derived << ", var : " << var
                  << ", var2: " << var2 << std::endl;
    }
};

void funcprint(Base *d) {
  Derived *temp = dynamic_cast<Derived*>(d);
  if(temp)  {
     d->do_something();
  }
  else {
```

```
      //Handle error
  }
}

Base *get_derived() {
  return new Derived;
}
```

Result Information

Group: Object Oriented
Language: C++
Default: On
Command-Line Syntax: INCOMPLETE_CLASS_PTR
Impact: Medium

See Also

Delete of void pointer | Find defects (-checkers) | MISRA C++:2008 Rule 5-2-4 | MISRA C++:2008 Rule 5-2-7 | MISRA C++:2008 Rule 5-2-8

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Copy constructor not called in initialization list

Copy constructor does not call copy constructors of some members or base classes

Description

This defect occurs when the copy constructor of a class does not call the *copy constructor* of the following in its initialization list:

- One or more of its members.
- Its base classes when applicable.

  The defect occurs even when a base class constructor is called instead of the base class copy constructor.

Risk

The calls to the copy constructors can be done only from the initialization list. If the calls are missing, it is possible that an object is only partially copied.

- If the copy constructor of a member is not called, it is possible that the member is not copied.
- If the copy constructor of a base class is not called, it is possible that the base class members are not copied.

Fix

If you want your copy constructor to perform a complete copy, call the copy constructor of all members and all base classes in its initialization list.

Examples

Base Class Copy Constructor Not Called

```
class Base {
public:
    Base();
    Base(int);
    Base(const Base&);
    virtual ~Base();
private:
    int ib;
};

class Derived:public Base {
public:
    Derived();
    ~Derived();
    Derived(const Derived& d): Base(), i(d.i) { }
private:
    int i;
};
```

In this example, the copy constructor of class `Derived` calls the default constructor, but not the copy constructor of class `Base`.

The defect appears on the `:` symbol in the copy constructor definition. Following are some tips for navigating in the source code:

- To navigate to the class definition, right-click a member that is initialized in the constructor. Select **Go To Definition**. In the class definition, you see the class members, including those members whose copy constructors are not called.
- To navigate to a base class definition, first navigate to the derived class definition. In the derived class definition, where the derived class inherits from a base class, right-click the base class name and select **Go To Definition**.

Correction — Call Base Class Copy Constructor

One possible correction is to call the copy constructor of class `Base` from the initialization list of the `Derived` class copy constructor.

```
class Base {
public:
    Base();
    Base(int);
    Base(const Base&);
    virtual ~Base();
private:
    int ib;
};

class Derived:public Base {
public:
    Derived();
    ~Derived();
    Derived(const Derived& d): Base(d), i(d.i) { }
private:
    int i;
};
```

Result Information

Group: Object oriented
Language: C++
Default: On
Command-Line Syntax: `MISSING_COPY_CTOR_CALL`
Impact: High

See Also

`Base class assignment operator not called|Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Copy operation modifying source operand

Copy operation modifies data member of source object

Description

This defect occurs when a copy constructor or copy assignment operator modifies a mutable data member of its source operand.

For instance, this copy constructor `A` modifies the data member `m` of its source operand `other`:

```
class A {
  mutable int m;

public:
 ...
  A(const A &other) : m(other.m) {
    other.m = 0; //Modification of source
  }
}
```

Risk

A copy operation with a copy constructor (or copy assignment operator):

```
className new_object = old_object; //Calls copy constructor of className
```

copies its source operand `old_object` to its destination operand `new_object`. After the operation, you expect the destination operand to be a copy of the unmodified source operand. If the source operand is modified during copy, this assumption is violated.

Fix

Do not modify the source operand in the copy operation.

If you are modifying the source operand in a copy constructor to implement a move operation, use a move constructor instead. Move constructors are defined in the C++11 standard and later.

Examples

Copy Constructor Modifying Source

```
#include <algorithm>
#include <vector>

class A {
  mutable int m;

public:
  A() : m(0) {}
  explicit A(int m) : m(m) {}

  A(const A &other) : m(other.m) {
```

```
    other.m = 0;
  }

  A& operator=(const A &other) {
    if (&other != this) {
      m = other.m;
      other.m = 0;
    }
    return *this;
  }

  int get_m() const { return m; }
};

void f() {
  std::vector<A> v{10};
  A obj(12);
  std::fill(v.begin(), v.end(), obj);
}
```

In this example, a vector of ten objects of type `A` is created. The `std::fill` function copies an object of type `A`, which has a data member with value 12, to each of the ten objects. After this operation, you might expect that all ten objects in the vector have a data member with value 12.

However, the first copy modifies the data member of the source to the value 0. The remaining nine copies copy this value. After the `std::fill` call, the first object in the vector has a data member with value 12 and the remaining objects have data members with value 0.

Correction — Use Move Constructor for Modifying Source

Do not modify data members of the source operand in a copy constructor or copy assignment operator. If you want your class to have a move operation, use a move constructor instead of a copy constructor.

In this corrected example, the copy constructor and copy assignment operator of class `A` do not modify the data member `m`. A separate move constructor modifies the source operand.

```
#include <algorithm>
#include <vector>

class A {
  int m;

public:
  A() : m(0) {}
  explicit A(int m) : m(m) {}

  A(const A &other) : m(other.m) {}
  A(A &&other) : m(other.m) { other.m = 0; }

  A& operator=(const A &other) {
    if (&other != this) {
      m = other.m;
    }
    return *this;
  }
```

```
  //Move constructor
  A& operator=(A &&other) {
    m = other.m;
    other.m = 0;
    return *this;
  }

  int get_m() const { return m; }
};

void f() {
  std::vector<A> v{10};
  A obj(12);
  std::fill(v.begin(), v.end(), obj);
}
```

Result Information

Group: Object Oriented
Language: C++
Default: On
Command-Line Syntax: `COPY_MODIFYING_SOURCE`
Impact: Medium

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

External Websites

Move constructors (C++11 and beyond)

Introduced in R2018b

Incompatible types prevent overriding

Derived class method hides a `virtual` base class method instead of overriding it

Description

This defect occurs when a derived class method has the same name and number of parameters as a `virtual` base class method but:

- The parameter lists differ in at least one parameter type.
- The parameter lists differ in the presence or absence of qualifiers such as `const`.

The derived class method hides the `virtual` base class method instead of overriding it.

Risk

- You might inadvertently hide the base class method instead of overriding it with the derived class method.
- If the base class method is hidden and you use a derived class object to call the method with the base class parameters, the derived class method is called instead. For the parameters whose types do not match the arguments that you pass, a cast takes place if possible. Otherwise, a compilation failure occurs.

Fix

To override a base class virtual method with a derived class method, declare the methods by using identical parameter lists. For instance, change the parameter type or add a `const` qualifier if required.

In C++11 and later, you can declare intended overriding methods in the derived class by using the specifier `override`. When you declare the derived class methods by using the specifier `override`, the compilation fails if the parameter lists of the base class method and the derived class method are different. The derived class methods cannot hide base class methods inadvertently and overriding of the base class virtual methods is ensured.

Otherwise, add the line `using` *`Base_class_name`*`::`*`method_name`* to the derived class declaration. You can then access the base class method using an object of the derived class.

Examples

`typedef` Causing Virtual Function Hiding in Derived Class

```
class Base {
public:
    Base();
    virtual ~Base();
    virtual void func(float i);
    virtual void funcp(float* i);
    virtual void funcr(float& i);
};
```

```
typedef double Float;

class Derived: public Base {
public:
    Derived();
    ~Derived();
    void func(Float i);
    void funcp(Float* i);
    void funcr(Float& i);
};
```

In this example, because of the statement `typedef double Float;`, the `Derived` class methods `func`, `funcp`, and `funcr` have `double` arguments while the `Base` class methods with the same name have `float` arguments. Therefore, you cannot access the `Base` class methods using a `Derived` class object.

The defect appears on the method that hides a base class method. To find which base class method is hidden:

1. Navigate to the base class definition. On the **Source** pane, right-click the base class name and select **Go To Definition**.
2. In the base class definition, identify the `virtual` method that has the same name as the derived class method name.

Correction — Unhide Base Class Method

One possible correction is to use the same argument type for the base and derived class methods to enable overriding. Otherwise, if you want to call the `Base` class methods with the `float` arguments using a `Derived` class object, add the line `using Base::`*`method_name`* to the `Derived` class declaration.

```
class Base {
public:
    Base();
    virtual ~Base();
    virtual void func(float i);
    virtual void funcp(float* i);
    virtual void funcr(float& i);
};

typedef double Float;

class Derived: public Base {
public:
    Derived();
    ~Derived();
    using Base::func;
    using Base::funcp;
    using Base::funcr;
    void func(Float i);
    void funcp(Float* i);
    void funcr(Float& i);
};
```

Correction — Specify Derived Class Method by using `override`

Another correction is to explicitly specify the derived class methods as overriding methods by using the specifier `override`. This way, it is clear that you intend to override the base class methods in the

derived class. If the overriding methods have different parameter lists than their base class counterparts, the code does not compile. As a result, the derived class methods cannot hide the base class methods.

```
class Base {
public:
    Base();
    virtual ~Base();
    virtual void func(float i);
    virtual void funcp(float* i);
    virtual void funcr(float& i);
};

typedef double Float;

class Derived: public Base {
public:
    Derived();
    ~Derived();
//   Compilation error
//  void func(Float i) override;
//  void funcp(Float* i) override;
//  void funcr(Float& i) override;

    void func(float i) override;
    void funcp(float* i) override;
    void funcr(float& i) override;
};
```

The commented out method definitions have different parameter lists compared to their base class counterparts. Because the derived class methods are declared by using the specifier `override`, the differing parameter lists do not hide the base class methods. Instead, the code fails to compile. Using the `override` specifier enforces the rule that virtual methods in base and derived classes must have identical parameter lists.

`const` Qualifier Missing in Derived Class Method

```
namespace Missing_Const {
class Base {
public:
    virtual void func(int) const ;
    virtual ~Base() ;
} ;

class Derived : public Base {
public:
    virtual void func(int) ;

} ;
}
```

In this example, `Derived::func` does not have a `const` qualifier but `Base::func` does. Therefore, `Derived::func` does not override `Base::func`.

Correction — Add `const` Qualifier to Derived Class Method

To enable overriding, add the `const` qualifier to the derived class method declaration.

```
namespace Missing_Const {
class Base {
public:
    virtual void func(int) const ;
    virtual ~Base() ;
} ;

class Derived : public Base {
public:
    virtual void func(int) const;

} ;
}
```

To avoid hiding base class methods or turning virtual methods into nonvirtual methods unintentionally:

- Declare virtual methods in the base class by using the specifier `virtual`.
- Declare virtual methods in a nonfinal derived base class by using the specifier `override`.
- Declare virtual methods in the final class by using the specifier `final`.

Value Instead of Reference in Derived Class Method

```
namespace Missing_Ref {

class Obj {
    int data;
};

class Base {
public:
    virtual void func(Obj& o);
    virtual ~Base() ;
} ;

class Derived : public Base {
public:
    virtual void func(Obj o) ;

} ;
}
```

In this example, `Derived::func` accepts an `Obj` parameter by value but `Base::func` accepts an `Obj` parameter by reference. Therefore, `Derived::func` does not override `Base::func`.

Correction — Use Reference for Parameter of Derived Class Method

To enable overriding, pass the derived class method parameter by reference.

```
namespace Missing_Ref {

class Obj {
    int data;
};

class Base {
public:
```

```
    virtual void func(Obj& o);
    virtual ~Base() ;
} ;

class Derived : public Base {
public:
    virtual void func(Obj& o) ;

} ;
}
```

To avoid hiding base class methods or turning virtual methods into nonvirtual methods unintentionally:

- Declare virtual methods in the base class by using the specifier `virtual`.
- Declare virtual methods in a nonfinal derived base class by using the specifier `override`.
- Declare virtual methods in the final class by using the specifier `final`.

Result Information

Group: Object oriented
Language: C++
Default: On
Command-Line Syntax: `VIRTUAL_FUNC_HIDING`
Impact: Medium

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Lambda used as typeid operand

`typeid` is used on lambda expression

Description

This defect occurs when you use `typeid` on a lambda expression.

Risk

According to the C++ Standard, the type of a lambda expression is a unique, unnamed class type. Because the type is unique, another variable or expression cannot have the same type. Use of `typeid` on a lambda expression indicates that you expect a second variable or expression to have the same type as the operand lambda expression. Using the type of a lambda expression in this way can lead to unexpected results.

`typeid` returns the data type of its operand. Typically the operator is used to compare the types of two variables. For instance:

```
(typeid(var1) == typeid(var2))
```

compares the types of `var1` and `var2`. This use does not apply to a lambda expression, which has a unique type.

Fix

Avoid using the `typeid` operator on lambda expressions.

Examples

Use of `typeid` on Lambda Expressions

```
#include <cstdint>
#include <typeinfo>

 void func()
 {
 auto lambdaFirst = []() -> std::int8_t { return 1; };
 auto lambdaSecond = []() -> std::int8_t { return 1; };

 if (typeid(lambdaFirst) == typeid(lambdaSecond))
     {
     // ...
     }
 }
```

The use of `typeid` on lambda expressions can lead to unexpected results. The comparison above is false even though `lambdaFirst` and `lambdaSecond` appear to have the same body.

Correction – Assign Lambda Expression to Function Object Before Using `typeid`

One possible correction is to assign the lambda expression to a function object and then use the `typeid` operator on the function objects for comparison.

```
#include <cstdint>
#include <functional>
#include <typeinfo>

 void func()
 {
 std::function<std::int8_t()> functionFirst = []() { return 1; };
 std::function<std::int8_t()> functionSecond = []() { return 1; };

 if (typeid(functionFirst) == typeid(functionSecond))
     {
     // ...
     }
 }
```

Result Information

Group: Object Oriented
Language: C++
Default: On
Command-Line Syntax: `LAMBDA_TYPE_MISUSE`
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019b

Member not initialized in constructor

Constructor does not initialize some members of a class

Description

This defect occurs when a class constructor has at least one execution path on which it does not initialize some data members of the class.

The defect does not appear in the following cases:

- Empty constructors.
- The non-initialized member is not used in the code.

Risk

The members that the constructor does not initialize can have unintended values when you read them later.

Initializing all members in the constructor makes it easier to use your class. If you call a separate method to initialize your members and then read them, you can avoid uninitialized values. However, someone else using your class can read a class member *before* calling your initialization method. Because a constructor is called when you create an object of the class, if you initialize all members in the constructor, they cannot have uninitialized values later on.

Fix

The best practice is to initialize all members in your constructor, preferably in an initialization list.

Examples

Non-Initialized Member

```
class MyClass {
public:
    explicit MyClass(int);
private:
    int _i;
    char _c;
};

MyClass::MyClass(int flag) {
    if(flag == 0) {
        _i = 0;
        _c = 'a';
    }
    else {
        _i = 1;
    }
}
```

In this example, if `flag` is not 0, the member `_c` is not initialized.

The defect appears on the closing brace of the constructor. Following are some tips for navigating in the source code:

- On the **Result Details** pane, see which members are not initialized.
- To navigate to the class definition, right-click a member that is initialized in the constructor. Select **Go To Definition**. In the class definition, you can see all the members, including those members that are not initialized in the constructor.

Correction — Initialize All Members on All Execution Paths

One possible correction is to initialize all members of the class `MyClass` for all values of `flag`.

```
class MyClass {
public:
    explicit MyClass(int);
private:
    int _i;
    char _c;
};

MyClass::MyClass(int flag) {
    if(flag == 0) {
        _i = 0;
        _c = 'a';
    }
    else {
        _i = 1;
        _c = 'b';
    }
}
```

Result Information

Group: Object oriented
Language: C++
Default: Off
Command-Line Syntax: `NON_INIT_MEMBER`
Impact: Medium
CWE ID: 456, 457, 908

See Also

`Copy constructor not called in initialization list|Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Missing explicit keyword

Constructor or user-defined conversion operator missing the `explicit` specifier

Description

This defect occurs when the declaration or in-class definition of a constructor or user-defined conversion operator does not use the `explicit` specifier. The `explicit` specifier prevents implicit conversion from a variable of another type to the current class type.

The defect applies to:

- One-parameter constructors.
- Constructors where all but one parameters have default values.

  For instance, `MyClass::MyClass(float f, bool b=true){}`.
- User-defined conversation operators.

  For instance, `operator int() {}` converts a variable of the current class type to an `int` variable.

Risk

If you do not declare a constructor or conversion operator `explicit`, compilers can perform implicit and often unintended type conversions to or from the class type with possibly unexpected results.

The implicit conversion using a constructor can occur, for instance, when a function accepts a parameter of the class type but you call the function with an argument of a different type. The call to `func` here causes an implicit conversion from type `int` to `myClass`:

```
class myClass {}{
  ...
  myClass(int) {...}
};
void func(myClass);
func(0);
```

The reverse implicit conversion can occur when using a user-defined conversion operator. For instance, you pass the class type as argument but the function has a parameter of a different type. The call to `func` here causes an implicit conversion from type `myClass` to `int`:

```
class myClass {} {
  ...
  operator int() {...}
};
myClass myClassObject;

void func(int) {...}
func(myClassObject);
```

Fix

For better readability of your code and to prevent implicit conversions, in the declaration or in-class definition of the constructor or conversion operator, place the `explicit` keyword before the constructor or operator name. You can then detect all implicit conversions as compilation errors and convert them to explicit conversions.

Examples

Missing `explicit` Keyword on Constructor

```
class MyClass {
public:
    MyClass(int val);
private:
    int val;
};

void func(MyClass);

void main() {
    MyClass MyClassObject(0);

    func(MyClassObject);   // No conversion
    func(MyClass(0));      // Explicit conversion
    func(0);               // Implicit conversion
}
```

In this example, the constructor of `MyClass` is not declared `explicit`. Therefore, the call `func(0)` can perform an implicit conversion from `int` to `MyClass`.

Correction — Use `explicit` Keyword

One possible correction is to declare the constructor of `MyClass` as `explicit`. If an operation in your code performs an implicit conversion, the compiler generates an error. Therefore, using the `explicit` keyword, you detect unintended type conversions in the compilation stage.

For instance, in function `main` below, if you add the statement `func(0);` that performs implicit conversion, the code does not compile.

```
class MyClass {
public:
    explicit MyClass(int val);
private:
    int val;
};

void func(MyClass);

void main() {
    MyClass MyClassObject(0);

    func(MyClassObject);   // No conversion
    func(MyClass(0));      // Explicit conversion
}
```

Incorrect Argument Order Preventable Through `explicit` Keyword

```
class Month {
    int val;
public:
    Month(int m): val(m) {}
    ~Month() {}
};

class Day {
    int val;
public:
    Day(int d): val(d) {}
    ~Day() {}
};

class Year {
    int val;
public:
    Year(int y): val(y) {}
    ~Year() {}
};

class Date {
    Month mm;
    Day dd;
    Year yyyy;
public:
    Date(const Month & m, const Day & d, const Year & y):mm(m), dd(d), yyyy(y) {}
};

void main() {
    Date(20,1,2000); //Implicit conversion, wrong argument order undetected
}
```

In this example, the constructors for classes `Month`, `Day` and `Year` do not have an `explicit` keyword. They allow implicit conversion from `int` variables to `Month`, `Day` and `Year` variables.

When you create a `Date` variable and use an incorrect argument order for the `Date` constructor, because of the implicit conversion, your code compiles. You might not detect that you have switched the month value and the day value.

Correction — Use `explicit` Keyword

If you use the `explicit` keyword for the constructors of classes `Month`, `Day` and `Year`, you cannot call the `Date` constructor with an incorrect argument order.

- If you call the `Date` constructor with `int` variables, your code does not compile because the `explicit` keyword prevents implicit conversion from `int` variables.
- If you call the `Date` constructor with the arguments explicitly converted to `Month`, `Day` and `Year`, and have the wrong argument order, your code does not compile because of the argument type mismatch.

```
class Month {
    int val;
public:
    explicit Month(int m): val(m) {}
```

```
    ~Month() {}
};

class Day {
    int val;
public:
    explicit Day(int d): val(d) {}
    ~Day() {}
};

class Year {
    int val;
public:
    explicit Year(int y): val(y) {}
    ~Year() {}
};

class Date {
    Month mm;
    Day dd;
    Year yyyy;
public:
    Date(const Month & m, const Day & d, const Year & y):mm(m), dd(d), yyyy(y) {}
};

void main() {
    Date(Month(1),Day(20),Year(2000));
    // Date(20,1,2000); - Does not compile
    // Date(Day(20), Month(1), Year(2000)); - Does not compile
}
```

Missing `explicit` Keyword on Conversion Operator

```
#include <cstdint>

class MyClass {
public:
    explicit MyClass(int32_t arg): val(arg) {};
    operator int32_t() const { return val; }
    explicit operator bool() const {
        if (val>0) {
          return true;
        }
        return false;
     }
private:
    int32_t val;
};

void useIntVal(int32_t);
void useBoolVal(bool);

void func() {
    MyClass MyClassObject{0};
    useIntVal(MyClassObject);
    useBoolVal(static_cast<bool>(MyClassObject));
}
```

In this example, the conversion operator `operator int32_t()` is not defined with the `explicit` specifier and allows implicit conversions. The conversion operator `operator bool()` is defined explicit.

When converting to a `bool` variable, for instance, in the call to `useBoolVal`, the `explicit` keyword in the conversion operator ensures that you have to perform an explicit conversion from the type `MyClass` to `bool`. There is no such requirement when converting to an `int32_t` variable. In the call to `useIntVal`, an implicit conversion is performed.

Result Information

Group: Object oriented
Language: C++
Default: Off
Command-Line Syntax: `MISSING_EXPLICIT_KEYWORD`
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Missing virtual inheritance

A base class is inherited virtually and nonvirtually in the same hierarchy

Description

This defect occurs when:

- A class is derived from multiple base classes, and some of those base classes are themselves derived from a common base class.

  For instance, a class `Final` is derived from two classes, `Intermediate_left` and `Intermediate_right`. Both `Intermediate_left` and `Intermediate_right` are derived from a common class, `Base`.
- At least one of the inheritances from the common base class is `virtual` and at least one is not `virtual`.

  For instance, the inheritance of `Intermediate_right` from `Base` is `virtual`. The inheritance of `Intermediate_left` from `Base` is not `virtual`.

Risk

If this defect appears, multiple copies of the base class data members appear in the final derived class object. To access the correct copy of the base class data member, you have to qualify the member and method name appropriately in the final derived class. The development is error-prone.

For instance, when the defect occurs, two copies of the base class data members appear in an object of class `Final`. If you do not qualify method names appropriately in the class `Final`, you can assign a value to a `Base` data member but not retrieve the same value.

- You assign the value using a `Base` method accessed through `Intermediate_left`. Therefore, you assign the value to one copy of the `Base` member.
- You retrieve the value using a `Base` method accessed through `Intermediate_right`. Therefore, you retrieve a different copy of the `Base` member.

Fix

Declare all the intermediate inheritances as `virtual` when a class is derived from multiple base classes that are themselves derived from a common base class.

If you indeed want multiple copies of the `Base` data members as represented in the intermediate derived classes, use aggregation instead of inheritance. For instance, declare two objects of class `Intermediate_left` and `Intermediate_right` in the `Final` class.

Examples

Missing Virtual Inheritance

```
#include <stdio.h>
class Base {
public:
```

```
    explicit Base(int i): m_b(i) {};
    virtual ~Base() {};
    virtual int get() const {
        return m_b;
    }
    virtual void set(int b) {
        m_b = b;
    }
private:
    int m_b;
};

class Intermediate_left: virtual public Base {
public:
    Intermediate_left():Base(0), m_d1(0) {};
private:
    int m_d1;
};

class Intermediate_right: public Base {
public:
    Intermediate_right():Base(0), m_d2(0) {};
private:
    int m_d2;
};

class Final: public Intermediate_left, Intermediate_right {
public:
    Final(): Base(0), Intermediate_left(), Intermediate_right() {};
    int get() const {
        return Intermediate_left::get();
    }
    void set(int b) {
        Intermediate_right::set(b);
    }
    int get2() const {
        return Intermediate_right::get();
    }
};

int main(int argc, char* argv[]) {
    Final d;
    int val = 12;
    d.set(val);
    int res = d.get();
    printf("d.get=%d\n",res);             // Result: d.get=0
    printf("d.get2=%d\n",d.get2());       // Result: d.get2=12
    return res;
}
```

In this example, `Final` is derived from both `Intermediate_left` and `Intermediate_right`. `Intermediate_left` is derived from `Base` in a non-`virtual` manner and `Intermediate_right` is derived from `Base` in a `virtual` manner. Therefore, two copies of the base class and the data member `m_b` are present in the final derived class,

Both derived classes `Intermediate_left` and `Intermediate_right` do not override the `Base` class methods `get` and `set`. However, `Final` overrides both methods. In the overridden `get` method,

it calls `Base::get` through `Intermediate_left`. In the overridden `set` method, it calls `Base::set` through `Intermediate_right`.

Following the statement `d.set(val)`, `Intermediate_right`'s copy of `m_b` is set to 12. However, `Intermediate_left`'s copy of `m_b` is still zero. Therefore, when you call `d.get()`, you obtain a value zero.

Using the `printf` statements, you can see that you retrieve a value that is different from the value that you set.

The defect appears in the final derived class definition and on the name of the class that are derived virtually from the common base class. Following are some tips for navigating in the source code:

- To find the definition of a class, on the **Source** pane, right-click the class name and select **Go To Definition**.
- To navigate up the class hierarchy, first navigate to the intermediate class definition. In the intermediate class definition, right-click a base class name and select **Go To Definition**.

Correction — Make Both Inheritances Virtual

One possible correction is to declare both the inheritances from `Base` as `virtual`.

Even though the overridden `get` and `set` methods in `Final` still call `Base::get` and `Base::set` through different classes, only one copy of `m_b` exists in `Final`.

```
#include <stdio.h>
class Base {
public:
    explicit Base(int i): m_b(i) {};
    virtual ~Base() {};
    virtual int get() const {
        return m_b;
    }
    virtual void set(int b) {
        m_b = b;
    }
private:
    int m_b;
};

class Intermediate_left: virtual public Base {
public:
    Intermediate_left():Base(0), m_d1(0) {};
private:
    int m_d1;
};

class Intermediate_right: virtual public Base {
public:
    Intermediate_right():Base(0), m_d2(0) {};
private:
    int m_d2;
};

class Final: public Intermediate_left, Intermediate_right {
public:
    Final(): Base(0), Intermediate_left(), Intermediate_right() {};
```

```
    int get() const {
        return Intermediate_left::get();
    }
    void set(int b) {
        Intermediate_right::set(b);
    }
    int get2() const {
        return Intermediate_right::get();
    }
};

int main(int argc, char* argv[]) {
    Final d;
    int val = 12;
    d.set(val);
    int res = d.get();
    printf("d.get=%d\n",res);             // Result: d.get=12
    printf("d.get2=%d\n",d.get2());       // Result: d.get2=12
    return res;
}
```

Result Information

Group: Object oriented
Language: C++
Default: Off
Command-Line Syntax: `MISSING_VIRTUAL_INHERITANCE`
Impact: Medium

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Object slicing

Derived class object passed by value to function with base class parameter

Description

This defect occurs when you pass a derived class object by value to a function, but the function expects a base class object as parameter.

Risk

If you pass a derived class object *by value* to a function, you expect the derived class copy constructor to be called. If the function expects a base class object as parameter:

1. The base class copy constructor is called.
2. In the function body, the parameter is considered as a base class object.

In C++, `virtual` methods of a class are resolved at run time according to the actual type of the object. Because of object slicing, an incorrect implementation of a `virtual` method can be called. For instance, the base class contains a `virtual` method and the derived class contains an implementation of that method. When you call the `virtual` method from the function body, the base class method is called, even though you pass a derived class object to the function.

Fix

One possible fix is to pass the object by reference or pointer. Passing by reference or pointer does not cause invocation of copy constructors. If you do not want the object to be modified, use a `const` qualifier with your function parameter.

Another possible fix is to overload the function with another function that accepts the derived class object as parameter.

Examples

Function Call Causing Object Slicing

```
#include <iostream>

class Base {
public:
    explicit Base(int b) {
        _b = b;
    }
    virtual ~Base() {}
    virtual int update() const;
protected:
    int _b;
};


class Derived: public Base {
public:
    explicit Derived(int b):Base(b) {}
```

```
    int update() const;
};

//Class methods definition

int Base::update() const {
    return (_b + 1);
}

int Derived::update() const {
    return (_b -1);
}


//Other function definitions
void funcPassByValue(const Base bObj) {
    std::cout << "Updated _b=" << bObj.update() << std::endl;
}

int main() {
    Derived dObj(0);
    funcPassByValue(dObj);       //Function call slices object
    return 0;
 }
```

In this example, the call `funcPassByValue(dObj)` results in the output `Updated _b=1` instead of the expected `Updated _b=-1`. Because `funcPassByValue` expects a `Base` object parameter, it calls the `Base` class copy constructor.

Therefore, even though you pass the `Derived` object `dObj`, the function `funcPassByValue` treats its parameter `b` as a `Base` object. It calls `Base::update()` instead of `Derived::update()`.

Correction — Pass Object by Reference or Pointer

One possible correction is to pass the `Derived` object `dObj` by reference or by pointer. In the following, corrected example, `funcPassByReference` and `funcPassByPointer` have the same objective as `funcPassByValue` in the preceding example. However, `funcPassByReference` expects a reference to a `Base` object and `funcPassByPointer` expects a pointer to a `Base` object.

Passing the `Derived` object `d` by a pointer or by reference does not slice the object. The calls `funcPassByReference(dObj)` and `funcPassByPointer(&dObj)` produce the expected result `Updated _b=-1`.

```
#include <iostream>

class Base {
public:
    explicit Base(int b) {
        _b = b;
    }
    virtual ~Base() {}
    virtual int update() const;
protected:
    int _b;
};
```

```
class Derived: public Base {
public:
    explicit Derived(int b):Base(b) {}
    int update() const;
};

//Class methods definition

int Base::update() const {
    return (_b + 1);
}

int Derived::update() const {
    return (_b -1);
}


//Other function definitions
void funcPassByReference(const Base& bRef) {
    std::cout << "Updated _b=" << bRef.update() << std::endl;
}

void funcPassByPointer(const Base* bPtr) {
    std::cout << "Updated _b=" << bPtr->update() << std::endl;
}

int main() {
    Derived dObj(0);
    funcPassByReference(dObj);       //Function call does not slice object
    funcPassByPointer(&dObj);       //Function call does not slice object
    return 0;
 }
```

Note If you pass by value, because a copy of the object is made, the original object is not modified. Passing by reference or by pointer makes the object vulnerable to modification. If you are concerned about your original object being modified, add a `const` qualifier to your function parameter, as in the preceding example.

Result Information

Group: Object oriented
Language: C++
Default: On
Command-Line Syntax: `OBJECT_SLICING`
Impact: High

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Operator new not overloaded for possibly overaligned class

Allocated storage might be smaller than object alignment requirement

Description

This defect occurs when you do not adequately overload operator `new/new[]` and you use this operator to create an object with an alignment requirement specified with `alignas`. The checker raises a defect for these versions of throwing and non-throwing operator `new/new[]`.

- `void* operator new(std::size_t size)`
- `void* operator new(std::size_t size, const std::nothrow_t&)`
- `void* operator new[](std::size_t size)`
- `void* operator new[](std::size_t size, const std::nothrow_t&)`

The use of `alignas` indicates that you do not expect the default operator `new/new[]` to satisfy the alignment requirement or the object, and that the object is possibly over aligned. A type is over aligned if you use `alignas` to make the alignment requirement of the type larger than `std::max_align_t`. For instance, `foo` is over aligned in this code snippet because its alignment requirement is 32 bytes, but `std::max_align_t` has an alignment of 16 bytes in most implementations.

```
struct alignas(32) foo {
  char elems[32];
}
```

Operator new not overloaded for possibly overaligned class raises no defect if you do not overload the operator `new/new[]` and you use version C++17 or later of the Standard. The default operator `new/new[]` in C++17 or later supports over alignment by passing the alignment requirement as an argument of type `std::align_val_t`, for instance `void* operator new(std::size_t size, std::align_val_t alignment)`.

Risk

The default operator `new/new[]` allocates storage with the alignment requirement of `std::align_val_t` at most. If you do not overload the operator when you create an object with over aligned type, the resulting object may be misaligned. Accessing this object might cause illegal access errors or abnormal program terminations.

Fix

If you use version C++14 or earlier of the Standard, pass the alignment requirement of over aligned types to the operator `new/new[]` by overloading the operator.

Examples

Allocated Memory Is Smaller Than Alignment Requirement of Type `foo`

```
#include <new>
```

```
#include <cstdlib>
#include <iostream>

struct alignas(64) foo {
    char elems[32];
};

foo*  func()
{
    foo*  bar = 0x0;
    try {
        bar =  new  foo ;
    } catch (...) { return nullptr; }
    delete bar;
}
```

In this example, structure `foo` is declared with an alignment requirement of 32 bytes. When you use the default operator `new` to create object `bar`, the allocated memory for `bar` is smaller than the alignment requirement of type `foo` and `bar` might be misaligned.

Correction — Define Overloaded Operator `new` to Handle Alignment Requirement of Type `foo`

One possible correction, if you use C11 `stdlib.h` or POSIX-C `malloc.h`, is to define an overloaded operator `new` that uses `aligned_alloc()` or `posix_memalign()` or to obtain storage with the correct alignment.

```
#include <new>
#include <cstdlib>
#include <iostream>

struct alignas(64) foo {
    char elems[32];
    static void* operator new (size_t nbytes)
    {
        if (void* p =
                ::aligned_alloc(alignof(foo), nbytes)) {
            return p;
        }
        throw std::bad_alloc();
    }
    static void operator delete(void *p) {
        free(p);
    }
};

foo*  func()
{
    foo*  bar = 0x0;
    try {
        bar =  new  foo ;
    } catch (...) { return nullptr; }
    delete bar;
}
```

Result Information

Group: Object Oriented
Language: C++
Default: On
Command-Line Syntax: `MISSING_OVERLOAD_NEW_FOR_ALIGNED_OBJ`
Impact: Medium

See Also

`Find defects (-checkers)|Missing overload of allocation or deallocation function`

Topics

“Interpret Polyspace Bug Finder Results”

“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019b

Partial override of overloaded virtual functions

Class overrides fraction of inherited virtual functions with a given name

Description

This defect occurs when:

- A base class has multiple `virtual` methods with the same name but different signatures (overloading).
- A class derived from the base class overrides at least one of those `virtual` methods, but not all of them.

Risk

The `virtual` methods that the derived class does not override are hidden. You cannot call those methods using an object of the derived class.

Fix

See if the overloads in the base class are required. If they are needed, possible solutions include:

- In your derived class, if you override one `virtual` method, override all `virtual` methods from the base class with the same name as that method.
- Otherwise, add the line `using` *`Base_class_name`*`::`*`method_name`* to the derived class declaration. In this way, you can call the base class methods using an object of the derived class.

Examples

Partial Override

```
class Base {
public:
    explicit Base(int b);
    virtual ~Base() {};
    virtual void set()           {
        _b = (int)0;
    };
    virtual void set(short i)    {
        _b = (int)i;
    };
    virtual void set(int i)      {
        _b = (int)i;
    };
    virtual void set(long i)     {
        _b = (int)i;
    };
    virtual void set(float i)    {
        _b = (int)i;
    };
    virtual void set(double i)   {
        _b = (int)i;
    };
```

```
private:
    int _b;
};

class Derived: public Base {
        public:
                Derived(int b, int d): Base(b), _d(d) {};
                void set(int i)    { Base::set(i); _d = (int)i; };
        private:
                int _d;
};
```

In this example, the class `Derived` overrides the function `set` that takes an `int` argument. It does not override other functions that have the same name `set` but take arguments of other types.

The defect appears on the derived class name in the derived class definition. To find which base class method is overridden:

1. Navigate to the base class definition. On the **Source** pane, right-click the base class name and select **Go To Definition**.
2. In the base class definition, identify the method that has the same name and signature as a derived class method name.

Correction — Unhide Base Class Method

One possible correction is add the line `using Base::set` to the `Derived` class declaration.

```
class Base {
public:
    explicit Base(int b);
    virtual ~Base() {};
    virtual void set()           {
        _b = (int)0;
    };
    virtual void set(short i)    {
        _b = (int)i;
    };
    virtual void set(int i)      {
        _b = (int)i;
    };
    virtual void set(long i)     {
        _b = (int)i;
    };
    virtual void set(float i)    {
        _b = (int)i;
    };
    virtual void set(double i)   {
        _b = (int)i;
    };
private:
    int _b;
};

class Derived: public Base {
        public:
                Derived(int b, int d): Base(b), _d(d) {};
                using Base::set;
```

```
                void set(int i)    { Base::set(i); _d = (int)i; };
        private:
                int _d;
};
```

Result Information

Group: Object oriented
Language: C++
Default: On
Command-Line Syntax: `PARTIAL_OVERRIDE`
Impact: Medium

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Return of non const handle to encapsulated data member

Method returns pointer or reference to internal member of object

Description

This defect occurs when:

- A class method returns a handle to a data member. Handles include pointers and references.
- The method is more accessible than the data member. For instance, the method has access specifier `public`, but the data member is `private` or `protected`.

Risk

The access specifier determines the accessibility of a class member. For instance, a class member declared with the `private` access specifier cannot be accessed outside a class. Therefore, nonmember, nonfriend functions cannot modify the member.

When a class method returns a handle to a less accessible data member, the member accessibility changes. For instance, if a `public` method returns a pointer to a `private` data member, the data member is effectively not `private` anymore. A nonmember, nonfriend function calling the `public` method can use the returned pointer to view and modify the data member.

Also, if you assign the pointer to a data member of an object to another pointer, when you delete the object, the second pointer can be left dangling. The second pointer points to the part of an object that does not exist anymore.

Fix

One possible fix is to avoid returning a handle to a data member from a class method. Return a data member by value so that a copy of the member is returned. Modifying the copy does not change the data member.

If you must return a handle, use a `const` qualifier with the method return type so that the handle allows viewing, but not modifying, the data member.

Examples

Return of Pointer to `private` Data Member

```
#include <string>
#define NUM_RECORDS 100

struct Date {
    int dd;
    int mm;
    int yyyy;
};
```

```
struct Period {
    Date startDate;
    Date endDate;
};

class DataBaseEntry {
private:
    std::string employeeName;
    Period employmentPeriod;
public:
    Period* getPeriod(void);
};

Period* DataBaseEntry::getPeriod(void) {
    return &employmentPeriod;
}


void use(Period*);
void reset(Period*);

int main() {
    DataBaseEntry dataBase[NUM_RECORDS];
    Period* tempPeriod;
    for(int i=0;i < NUM_RECORDS;i++) {
        tempPeriod = dataBase[i].getPeriod();
        use(tempPeriod);
        reset(tempPeriod);
    }
    return 0;
}

void reset(Period* aPeriod) {
       aPeriod->startDate.dd = 1;
       aPeriod->startDate.mm = 1;
       aPeriod->startDate.yyyy = 2000;
}
```

In this example, `employmentPeriod` is `private` to the class `DataBaseEntry`. It is therefore immune from modification by nonmember, nonfriend functions. However, returning a pointer to `employmentPeriod` breaks this encapsulation. For instance, the nonmember function `reset` modifies the member `startDate` of `employmentPeriod`.

Correction: Return Member by Value

One possible correction is to return the data member `employmentPeriod` by value instead of pointer. Modifying the return value does not change the data member because the return value is a copy of the data member.

```
#include <string>
#define NUM_RECORDS 100

struct Date {
    int dd;
    int mm;
    int yyyy;
};
```

```
struct Period {
    Date startDate;
    Date endDate;
};

class DataBaseEntry {
private:
    std::string employeeName;
    Period employmentPeriod;
public:
    Period getPeriod(void);
};

Period DataBaseEntry::getPeriod(void) {
    return employmentPeriod;
}


void use(Period*);
void reset(Period*);

int main() {
    DataBaseEntry dataBase[NUM_RECORDS];
    Period tempPeriodVal;
    Period* tempPeriod;
    for(int i=0;i < NUM_RECORDS;i++) {
        tempPeriodVal = dataBase[i].getPeriod();
        tempPeriod = &tempPeriodVal;
        use(tempPeriod);
        reset(tempPeriod);
    }
    return 0;
}

void reset(Period* aPeriod) {
       aPeriod->startDate.dd = 1;
       aPeriod->startDate.mm = 1;
       aPeriod->startDate.yyyy = 2000;
}
```

Result Information

Group: Object oriented
Language: C++
Default: Off
Command-Line Syntax: `BREAKING_DATA_ENCAPSULATION`
Impact: Medium
CWE ID: 375, 767

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Self assignment not tested in operator

Copy assignment operator does not test for self-assignment

Description

This defect occurs when you do not test if the argument to the copy assignment operator of an object is the object itself.

Risk

Self-assignment causes unnecessary copying. Though it is unlikely that you assign an object to itself, because of aliasing, you or users of your class cannot always detect a self-assignment.

Self-assignment can cause subtle errors if a data member is a pointer and you allocate memory dynamically to the pointer. In your copy assignment operator, you typically perform these steps:

1. Deallocate the memory originally associated with the pointer.

   ```
   delete ptr;
   ```

2. Allocate new memory to the pointer. Initialize the new memory location with contents obtained from the operator argument.

   ```
    ptr = new ptrType(*(opArgument.ptr));
   ```

If the argument to the operator, `opArgument`, is the object itself, after your first step, the pointer data member in the operator argument, `opArgument.ptr`, is not associated with a memory location. `*opArgument.ptr` contains unpredictable values. Therefore, in the second step, you initialize the new memory location with unpredictable values.

Fix

Test for self-assignment in the copy assignment operator of your class. Only after the test, perform the assignments in the copy assignment operator.

Examples

Missing Test for Self-Assignment

```
class MyClass1 { };
class MyClass2 {
public:
    MyClass2() : p_(new MyClass1()) { }
    MyClass2(const MyClass2& f) : p_(new MyClass1(*f.p_)) { }
    ~MyClass2() {
        delete p_;
    }
    MyClass2& operator= (const MyClass2& f)
    {
        delete p_;
        p_ = new MyClass1(*f.p_);
        return *this;
    }
```

```
private:
    MyClass1* p_;
};
```

In this example, the copy assignment operator in `MyClass2` does not test for self-assignment. If the parameter `f` is the current object, after the statement `delete p_`, the memory allocated to pointer `f.p_` is also deallocated. Therefore, the statement `p_ = new MyClass1(*f.p_)` initializes the memory location that `p_` points to with unpredictable values.

Correction — Test for Self-Assignment

One possible correction is to test for self-assignment in the copy assignment operator.

```
class MyClass1 { };
class MyClass2 {
public:
    MyClass2() : p_(new MyClass1()) { }
    MyClass2(const MyClass2& f) : p_(new MyClass1(*f.p_)) { }
    ~MyClass2() {
        delete p_;
    }
    MyClass2& operator= (const MyClass2& f)
    {
        if(&f != this) {
           delete p_;
           p_ = new MyClass1(*f.p_);
        }
        return *this;
    }
private:
    MyClass1* p_;
};
```

Result Information

Group: Object oriented
Language: C++
Default: Off
Command-Line Syntax: `MISSING_SELF_ASSIGN_TEST`
Impact: Medium

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Performance Defects

Const parameter values may cause unnecessary data copies

Const parameter values may prevent a move operation resulting in a more performance-intensive copy operation

Description

This defect occurs when `const` objects as function parameters may prevent a move operation resulting in a more performance-intensive copy operation.

The checker does not check if a move operation is possible in a given function call. The checker simply highlights `const` function parameters that have class types with a nontrivial copy operation and a move operation. You can determine for yourself if the parameter can be moved to the called function.

Risk

If the function argument is an rvalue, the resources associated with the argument are no longer required and can be *moved* to parameters in the called function. Compilers ensure that the move operation is used in this situation since they are generally less expensive than copy operations. If you use a `const` object as function parameter, you explicitly prevent this compiler optimization.

Fix

If you think that the parameter can be moved to the called function, remove the `const` qualifier from the flagged function parameter.

Examples

`const` Parameter Value Preventing Move Operation

```
#include <string>

std::string getStringFromUser() {
    //Get a string of arbitrary length
}

void countWordsInString(const std::string str) {
    //Count number of words in string
}

void main() {
    std::string aString = getStringFromUser();
    std::string anotherString = getStringFromUser();

    std::string joinedString = aString + anotherString;

    countWordsInString(joinedString);
    countWordsInString(aString + anotherString);exit
}
```

In this example, the checker flags the `const str::string` parameter `str`. In situations where a move operation is possible, for example in the call:

```
countWordsInString(aString + anotherString);
```

the `const` parameter forces a copy operation, which can be significantly more expensive.

Check Information

Group: Performance
Language: C++
Default: Off
Command-Line Syntax: `CONST_PARAMETER_VALUE`
Impact: Low

See Also

`Const return values may cause unnecessary data copies` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Const return values may cause unnecessary data copies

Const return values may prevent a move operation resulting in a more performance-intensive copy operation

Description

This defect occurs when `const` objects as return values may prevent a move operation resulting in a more performance-intensive copy operation.

The checker does not check if a move operation is possible for any calling function. The checker simply highlights `const` function return values that have class types with a nontrivial copy operation and a move operation.

Risk

The resources associated with the function return value are no longer required and can be moved to objects in the calling function. Compilers ensure that the move operation is used in this situation since they are generally less expensive than copy operations. If you use a `const` object as return value, you explicitly prevent this compiler optimization.

In addition, the calling function can store the return value in a non-`const` object. The `const`-ness of the return value does not prevent any operation on the non-`const` object.

Fix

Remove `const` qualifiers from function return values.

Examples

`const` Return Value Preventing Move Operation

```
#include <string>

class stringPair {
    std::string str1;
    std::string str2;

    public:
    stringPair& operator=(const stringPair & aPair){
        if(&aPair != this) {
            str1 = aPair.str1;
            str2 = aPair.str2;
        }
        return *this;
    }

    const std::string getJoinedString(void) {
        return (str1 + str2);
    }
};
```

In this example, the `const` specifier on the return value of `getJoinedString` forces a copy operation instead of move operations.

Check Information

Group: Performance
Language: C++
Default: Off
Command-Line Syntax: `CONST_RETURN_VALUE`
Impact: Low

See Also

`Const parameter values may cause unnecessary data copies` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Empty destructors may cause unnecessary data copies

User-declared empty destructors prevent autogeneration of move constructors and move assignment operators

Description

This defect occurs when a class definition contains a user-declared destructor with an empty or `=default` definition and does not declare both a move constructor and move assignment operator. For instance:

```
class aClass
{
public:
    ~aClass() noexcept
     {} // Empty body
};
class bClass
{
public:
    ~bClass() = default;
};
```

The destructors above are exactly the same as the compiler provided version, but they prevent automatic generation of the move operators. As a result, the class type is not movable. The defect appears even if you declare the destructors with `=delete`.

An empty destructor is not flagged if:

- The destructor is private or protected.
- The destructor is declared virtual or final.
- The destructor overrides a base class pure virtual destructor.

Risk

Instances of this class might be unnecessarily copied in situations where a move operation would have been possible. Copy operations are more expensive than move operations and might impact performance.

Fix

Try one of these solutions:

- Remove the empty destructor if possible. If a class does not have a destructor, the compiler generates a destructor, which is essentially the empty destructor that you explicitly declared.
- If you cannot remove the destructor, add an explicit move constructor and move assignment operator to the class definition. Use the `=default` syntax to clarify that the compiler definitions of move constructors and move assignment operators are used.

  ```
  class aClass
  {
  ```

```
public:
     ~aClass() noexcept = default;
     aClass(aClass&& ) = default;
     aClass& operator=(aClass&& ) = default;
};
```

Check Information

Group: Performance
Language: C++
Default: Off
Command-Line Syntax: `EMPTY_DESTRUCTOR_DEFINED`
Impact: High

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Inefficient string length computation

String length calculated by using string length functions on return from `std::basic_string::c_str()` instead of using `std::basic_string::length()`

Description

This defect occurs when the length of a `std::basic_string` string is calculated by using string length functions on the pointer returned from `std::basic_string::c_str()` instead of using the method `std::basic_string::length()`.

The checker flags string length functions such as `strlen`, `wcslen` and `char_traits::length`.

Risk

`std::basic_string::c_str()` returns a pointer to a null-terminated character array that stores the same data as the data stored in the string. Using a string length function such as `strlen` on this character array is expected to return the string length. This approach might seem superficially equivalent to using the `std::basic_string::length()` method for the string length.

However, the function `strlen(str)` is of linear complexity O(N) where N is the length of string `str`. If `str` is of type `std::basic_string`, this complexity is unnecessary since calling the `std::basic_string::length()` method returns the length more efficiently (with complexity O(1)).

Fix

If a string is of type `std::basic_string`, to get the string length, instead of using string length functions such as `strlen`, for instance:

```
std::string s;
auto len = strlen(s.ctr());
```

use the `std::basic_string::length()` method, for instance:

```
std::string s;
auto len = s.length();
```

Check Information

Group: Performance
Language: C++
Default: Off
Command-Line Syntax: `INEFFICIENT_BASIC_STRING_LENGTH`
Impact: High

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

std::endl may cause an unnecessary flush

std::endl is used instead of the more efficient \n

Description

This defect flags uses of std::endl in I/O operations and allows you to use the more efficient alternative, \n.

Risk

std::endl inserts a newline (\n) followed by a flush operation. For instance:

```
std::cout << "Some content" << std::endl;
```

is equivalent to:

```
std::cout << "Some content" << '\n' << std::flush;
```

The implicit flush operation might not be necessary or intended. If your program has many I/O operations that use std::endl, the implicit flush operation can significantly reduce program performance. Since the flush operation is implicit, in case of a performance issue, it will be difficult to track the root cause of the issue.

Fix

Use \n to enter a newline wherever possible.

If you require a flush operation, instead of std::endl, use \n followed by an explicit flush operation, for instance:

```
std::cout << "Some content" << '\n' << std::flush;
```

In this case, the analysis considers your use of a flush operation as deliberate and does not flag the use.

Examples

Possible Performance Impact from std::endl Use

```
#include <fstream>
using namespace std;

int main()
{
  ofstream aFile("file.txt");
  for ( int i = 0; i < 100000; i++) {
    aFile << "Hello World " << std::endl ;
  }
  aFile.close();
  return 0;
}
```

In this example, an `std::endl` is used in a loop during a write operation on a file. Since the loop has 100000 iterations, the slight delay from each implicit flush operation can add up to a significant reduction of performance.

Use \n and Avoid Flush

In a loop with several iterations, avoid the performance reduction in I/O operations by using `\n` instead of `std::endl`.

```
#include <fstream>
using namespace std;

int main()
{
  ofstream aFile("file.txt");
  for ( int i = 0; i < 100000; i++) {
    aFile << "Hello World \n" ;
  }
  aFile.close();
  return 0;
}
```

Check Information

Group: Performance
Language: C++
Default: Off
Command-Line Syntax: `STD_ENDL_USE`
Impact: High

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Resource Management Defects

Closing a previously closed resource

Function closes a previously closed stream

Description

This defect occurs when a function attempts to close a stream that was closed earlier in your code and not reopened later.

Risk

The standard states that the value of a `FILE*` pointer is indeterminate after you close the stream associated with it. Performing the close operation on the `FILE*` pointer again can cause unwanted behavior.

Fix

Remove the redundant close operation.

Examples

Closing Previously Closed Resource

```
#include <stdio.h>

void func(char* data) {
    FILE* fp = fopen("file.txt", "w");
    if(fp!=NULL) {
        if(data)
            fputc(*data,fp);
        else
            fclose(fp);
    }
    fclose(fp);
}
```

In this example, if `fp` is not `NULL` and `data` is `NULL`, the `fclose` operation occurs on `fp` twice in succession.

Correction — Remove Close Operation

One possible correction is to remove the last `fclose` operation. To avoid a resource leak, you must also place an `fclose` operation in the `if(data)` block.

```
#include <stdio.h>

void func(char* data) {
    FILE* fp = fopen("file.txt", "w");
    if(fp!=NULL) {
        if(data) {
            fputc(*data,fp);
            fclose(fp);
        }
        else
```

```
            fclose(fp);
    }
}
```

Result Information

Group: Resource management
Language: C | C++
Default: On
Command-Line Syntax: DOUBLE_RESOURCE_CLOSE
Impact: High
CWE ID: 672, 826, 910

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Opening previously opened resource

Opening an already opened file

Description

This defect occurs when a file handling function such as `fopen` opens a file that was previously opened and not closed subsequently.

Risk

If you open a resource multiple times, you can encounter:

- A race condition when accessing the file.
- Undefined or unexpected behavior for that file.
- Portability issues when you run your program on different targets.

Fix

Once a resource is open, close the resource before reopening.

Examples

File Reopened With New Permissions

```
#include <stdio.h>
const char* logfile = "my_file.log";

void doubleresourceopen()
{
    FILE* fpa = fopen(logfile, "w");
    if (fpa == NULL) {
        return;
    }
    (void)fprintf(fpa, "Writing");
    FILE* fpb = fopen(logfile, "r");
    (void)fclose(fpa);
    (void)fclose(fpb);
}
```

In this example, a `logfile` is opened in the first line of this function with write privileges. Halfway through the function, the `logfile` is opened again with read privileges.

Correction — Close Before Reopening

One possible correction is to close the file before reopening the file with different privileges.

```
#include <stdio.h>
const char* logfile = "my_file.log";

void doubleresourceopen()
{
    FILE* fpa = fopen(logfile, "w");
```

```
    if (fpa == NULL) {
        return;
    }
    (void)fprintf(fpa, "Writing");
    (void)fclose(fpa);
    FILE* fpb = fopen(logfile, "r");
    (void)fclose(fpb);
}
```

Result Information

Group: Resource management
Language: C | C++
Default: On
Command-Line Syntax: `DOUBLE_RESOURCE_OPEN`
Impact: Medium
CWE ID: 362, 413, 675

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2016b

Resource leak

File stream not closed before `FILE` pointer scope ends or pointer is reassigned

Description

This defect occurs when you open a file stream by using a `FILE` pointer but do not close it before:

- The end of the pointer's scope.
- Assigning the pointer to another stream.

Risk

If you do not release file handles explicitly as soon as possible, a failure can occur due to exhaustion of resources.

Fix

Close a `FILE` pointer before the end of its scope, or before you assign the pointer to another stream.

Examples

`FILE` Pointer Not Released Before End of Scope

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

In this example, the file pointer `fp1` is pointing to a file `data1.txt`. Before `fp1` is explicitly dissociated from the file stream of `data1.txt`, it is used to access another file `data2.txt`.

Correction — Release `FILE` Pointer

One possible correction is to explicitly dissociate `fp1` from the file stream of `data1.txt`.

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );
    fclose(fp1);

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
```

```
    fclose ( fp1 );
}
```

Result Information

Group: Resource management
Language: C | C++
Default: On
Command-Line Syntax: RESOURCE_LEAK
Impact: High
CWE ID: 772

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Use of previously closed resource

Function operates on a previously closed stream

Description

This defect occurs when a function operates on a stream that you closed earlier in your code.

Risk

The standard states that the value of a `FILE*` pointer is indeterminate after you close the stream associated with it. Operations using the `FILE*` pointer can produce unintended results.

Fix

One possible fix is to close the stream only at the end of operations. Another fix is to reopen the stream before using it again.

Examples

Use of `FILE*` Pointer After Closing Stream

```
#include <stdio.h>

void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fclose(fp);
        fprintf(fp,"text");
    }
}
```

In this example, `fclose` closes the stream associated with `fp`. When you use `fprintf` on `fp` after `fclose`, the **Use of previously closed resource** defect appears.

Correction — Close Stream After All Operations

One possible correction is to reverse the order of the `fprintf` and `fclose` operations.

```
#include <stdio.h>

void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fprintf(fp,"text");
        fclose(fp);
    }
}
```

Result Information

Group: Resource management
Language: C | C++
Default: On
Command-Line Syntax: `CLOSED_RESOURCE_USE`
Impact: High
CWE ID: 672, 826, 910

See Also

`Find defects (-checkers)|MISRA C:2012 Rule 22.6`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Writing to read-only resource

File initially opened as read only is modified

Description

This defect occurs when you attempt to write to a file that you have opened earlier in read-only mode.

For instance, you open a file using `fopen` with the access mode argument `r`. You write to that file with a function in the `fprintf` family.

Risk

Writing to a read-only file causes undefined behavior.

Fix

If you want to write to the file, open the file in a mode that is suitable for writing.

Examples

Writing to Read-Only File

```
#include <stdio.h>

void func(void) {
    FILE* fp ;

    fp = fopen("file.txt", "r");
    fprintf(fp, "Some data");
    fclose(fp);
}
```

In this example, the file `file.txt` is opened in read-only mode. When the `FILE` pointer associated with `file.txt` is used as an argument of `fprintf`, a **Writing to read-only resource** defect occurs.

Correction — Open File as Writable

One possible correction is to use the access specifier `"a"` instead of `"r"`. `file.txt` is now open for output at the end of the file.

```
#include <stdio.h>

void func(void) {
    FILE* fp ;

    fp = fopen("file.txt", "a");
    fprintf(fp, "Some data");
    fclose(fp);
}
```

Result Information

Group: Resource management

Language: C | C++
Default: On
Command-Line Syntax: `READ_ONLY_RESOURCE_WRITE`
Impact: High

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Good Practice Defects

Ambiguous declaration syntax

Declaration syntax can be interpreted as object declaration or part of function declaration

Description

This defect occurs when it is not clear from a declaration whether an object declaration or function/parameter declaration is intended. The ambiguity is often referred to as most vexing parse.

For instance, these declarations are ambiguous:

- `ResourceType aResource();`

  It is not immediately clear if `aResource` is a function returning a variable of type `ResourceType` or an object of type `ResourceType`.
- `TimeKeeper aTimeKeeper(Timer());`

  It is not immediately clear if `aTimeKeeper` is an object constructed with an unnamed object of type `Timer` or a function with an unnamed function pointer type as parameter. The function pointer refers to a function with no argument and return type `Timer`.

Risk

In case of an ambiguous declaration, the C++ Standard chooses a specific interpretation of the syntax. For instance:

- `ResourceType aResource();`

  is interpreted as a declaration of a function `aResource`.
- `TimeKeeper aTimeKeeper(Timer());`

  is interpreted as a declaration of a function `aTimeKeeper` with an unnamed parameter of function pointer type.

If you or another developer or code reviewer expects a different interpretation, the results can be unexpected.

For instance, later you might face a compilation error that is difficult to understand. Since the default interpretation indicates a function declaration, if you use the function as an object, compilers might report a compilation error. The compilation error indicates that a conversion from a function to an object is being attempted without a suitable constructor.

Fix

Make the declaration unambiguous. For instance, fix these ambiguous declarations as follows:

- `ResourceType aResource();`

  Object declaration:

  If the declaration refers to an object initialized with the default constructor, rewrite it as:

  ```
  ResourceType aResource;
  ```

prior to C++11, or as:

```
ResourceType aResource{};
```

after C++11.

Function declaration:

If the declaration refers to a function, use a typedef for the function.

```
typedef ResourceType(*resourceFunctionType)();
resourceFunctionType aResource;
```

- `TimeKeeper aTimeKeeper(Timer());`

  Object declaration:

  If the declaration refers to an object `aTimeKeeper` initialized with an unnamed object of class `Timer`, add an extra pair of parenthesis:

  ```
  TimeKeeper aTimeKeeper( (Timer()) );
  ```

  prior to C++11, or use braces:

  ```
  TimeKeeper aTimeKeeper{Timer{}};
  ```

  after C++11.

  Function declaration:

  If the declaration refers to a function `aTimeKeeper` with a unnamed parameter of function pointer type, use a named parameter instead.

  ```
  typedef Timer(*timerType)();
  TimeKeeper aTimeKeeper(timerType aTimer);
  ```

Examples

Function or Object Declaration

```
class ResourceType {
      int aMember;
    public:
      int getMember();
};

void getResource() {
    ResourceType aResource();
}
```

In this example, `aResource` might be used as an object but the declaration syntax indicates a function declaration.

Correction — Use {} for Object Declaration

One possible correction (after C++11) is to use braces for object declaration.

```
class ResourceType {
```

```
      int aMember;
    public:
      int getMember();
};

void getResource() {
    ResourceType aResource{};
}
```

Unnamed Object or Unnamed Function Parameter Declaration

```
class MemberType {};

class ResourceType {
      MemberType aMember;
    public:
      ResourceType(MemberType m) {aMember = m;}
      int getMember();
};

void getResource() {
    ResourceType aResource(MemberType());
}
```

In this example, `aResource` might be used as an object initialized with an unnamed object of type `MemberType` but the declaration syntax indicates a function with an unnamed parameter of function pointer type. The function pointer points to a function with no arguments and type `MemberType`.

Correction — Use {} for Object Declaration

One possible correction (after C++11) is to use braces for object declaration.

```
class MemberType {};

class ResourceType {
      MemberType aMember;
    public:
      ResourceType(MemberType m) {aMember = m;}
      int getMember();
};

void getResource() {
    ResourceType aResource{MemberType()};
}
```

Unnamed Object or Named Function Parameter Declaration

```
class Integer {
       int aMember;
    public:
       Integer(int d) {aMember = d;}
       int getMember();
};

int aInt = 0;
Integer aInteger(Integer(aInt));
```

In this example, `aInteger` might be an object constructed with an unnamed object `Integer(aInt)` (an object of class `Integer` which itself is constructed using the variable `aInt`). However, the declaration syntax indicates that `aInteger` is a function with a named parameter `aInt` of type `Integer` (the superfluous parenthesis is ignored).

Correction — Use of {} for Object Declaration

One possible correction (after C++11) is to use {} for object declaration.

```
class Integer {
       int aMember;
    public:
       Integer(int d) {aMember = d;}
       int getMember();
};

int aInt = 0;
Integer aInteger{Integer{aInt}};
```

Correction — Remove Superfluous Parenthesis for Named Parameter Declaration

If `aInteger` is a function with a named parameter `aInt`, remove the superfluous () around `aInt`.

```
class Integer {
       int aMember;
    public:
       Integer(int d) {aMember = d;}
       int getMember();
};

Integer aInteger(Integer aInt);
```

Result Information

Group: Good practice
Language: C++
Default: Off
Command-Line Syntax: `MOST_VEXING_PARSE`
Impact: Low

See Also

`Find defects (-checkers)` | `Improper array initialization` | `Non-initialized variable` | `Variable shadowing` | `Write without a further read`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019a

Bitwise and arithmetic operation on the same data

Statement with mixed bitwise and arithmetic operations

Description

This defect occurs when bitwise and arithmetic operations are performed in the same expression.

Risk

Mixed bitwise and arithmetic operations *do* compile. However, the size of integer types affects the result of these mixed operations. For instance, the arithmetic equivalent of a left shift (<<) by a certain number of bits depends on the number of bits in the variable being shifted and therefore on the internal representation of its data type. With a mix of bitwise and arithmetic operations, the same expression can produce different results on different targets.

Mixed operations also reduce readability and maintainability.

Fix

Separate bitwise and arithmetic operations, or use only one type of operation per statement.

Examples

Shift and Addition

```
unsigned int bitwisearithmix()
{
    unsigned int var = 50;
    var += (var << 2) + 1;
    return var;
}
```

This example shows bitwise and arithmetic operations on the variable `var`. `var` is shifted by two (bitwise), then increased by 1 and added to itself (arithmetic).

Correction — Arithmetic Operations Only

You can reduce this expression to arithmetic-only operations: `var + (var << 2)` is equivalent to `var * 5`.

```
unsigned int bitwisearithmix()
{
    unsigned int var = 50;
    var = var * 5 +1;
    return var;
}
```

Result Information

Group: Good Practice
Language: C | C++
Default: Off

Command-Line Syntax: `BITWISE_ARITH_MIX`
Impact: Low
CWE ID: 710

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2016b

C++ reference to const-qualified type with subsequent modification

Reference to `const`-qualified type is subsequently modified

Description

This defect occurs when a variable that refers to a `const`-qualified type is modified after declaration.

For instance, in this example, `refVal` has a type `const int &`, but its value is modified in a subsequent statement.

```
using constIntRefType = const int &;
void func(constIntRefType refVal, int val){
   ...
   refVal = val; //refVal is modified
   ...
}
```

Risk

The `const` qualifier on a reference type implies that a variable of the type is initialized at declaration and will not be subsequently modified.

Compilers can detect modification of references to `const`-qualified types as a compilation error. If the compiler does not detect the error, the behavior is undefined.

Fix

Avoid modification of `const`-qualified reference types. If the modification is required, remove the `const` qualifier from the reference type declaration.

Examples

Modification of `const`-qualified Reference Types

```
typedef const int cint;
typedef cint& ref_to_cint;

void func(ref_to_cint refVal, int initVal){
   refVal = initVal;
}
```

In this example, `ref_to_cint` is a reference to a `const`-qualified type. The variable `refVal` of type `ref_to_cint` is supposed to be initialized when `func` is called and not modified subsequently. The modification violates the contract implied by the `const` qualifier.

Correction — Avoid Modification of `const`-qualified Reference Types

One possible correction is to avoid the `const` in the declaration of the reference type.

```
typedef int& ref_to_int;
```

```
void func(ref_to_int refVal, int initVal){
   refVal = initVal;
}
```

Result Information

Group: Good practice
Language: C++
Default: Off
Command-Line Syntax: `WRITE_REFERENCE_TO_CONST_TYPE`
Impact: Low

See Also

`C++ reference type qualified with const or volatile|Find defects (-checkers)|Qualifier removed in conversion|Writing to const qualified object`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019a

C++ reference type qualified with const or volatile

Reference type declared with a redundant `const` or `volatile` qualifier

Description

This defect occurs when a variable with reference type is declared with the `const` or `volatile` qualifier, for instance:

```
char &const c;
```

Risk

The C++14 Standard states that `const` or `volatile` qualified references are ill formed (unless they are introduced through a `typedef`, in which case they are ignored). For instance, a reference to one variable cannot be made to refer to another variable. Therefore, using the `const` qualifier is not required for a variable with a reference type.

Often the use of these qualifiers indicate a coding error. For instance, you meant to declare a reference to a `const`-qualified type:

```
char const &c;
```

but instead declared a `const`-qualified reference:

```
char &const c;
```

If your compiler does not detect the error, you can see unexpected results. For instance, you might expect `c` to be immutable but see a different value of `c` compared to its value at declaration.

Fix

See if the `const` or `volatile` qualifier is incorrectly placed. For instance, see if you wanted to refer to a `const`-qualified type and entered:

```
char &const c;
```

instead of:

```
char const &c;
```

If the qualifier is incorrectly placed, fix the error. Place the `const` or `volatile`qualifier before the & operator. Otherwise, remove the redundant qualifier.

Examples

`const`-Qualified Reference Type

```
int func (int &const iRef) {
    iRef++;
    return iRef%2;
}
```

In this example, `iRef` is a `const`-qualified reference type. Since `iRef` cannot refer to another variable, the `const` qualifier is redundant.

Correction — Remove `const` Qualifier

Remove the redundant `const` qualifier. Since `iRef` is modified in `func`, it is not meant to refer to a `const`-qualified variable. Moving the `const` qualifier before & will cause a compilation error.

```
int func (int &iRef) {
    iRef++;
    return iRef%2;
}
```

Correction — Fix Placement of `const` Qualifier

If you do not identify to modify `iRef` in `func`, declare `iRef` as a reference to a `const`-qualified variable. Place the `const` qualifier before the & operator. Make sure you do not modify `iRef` in `func`.

```
int func (int const &iRef) {
    return (iRef+1)%2;
}
```

Result Information

Group: Good practice
Language: C++
Default: Off
Command-Line Syntax: `CV_QUALIFIED_REFERENCE_TYPE`
Impact: Low

See Also

`C++ reference to const-qualified type with subsequent modification` | `Find defects (-checkers)` | `Qualifier removed in conversion` | `Unreliable cast of function pointer` | `Unreliable cast of pointer` | `Writing to const qualified object`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019a

Delete of void pointer

`delete` operates on a `void*` pointer pointing to an object

Description

This defect occurs when the `delete` operator operates on a `void*` pointer.

Risk

Deleting a `void*` pointer is undefined according to the C++ Standard.

If the object is of type `MyClass` and the `delete` operator operates on a `void*` pointer pointing to the object, the `MyClass` destructor is not called.

If the destructor contains cleanup operations such as release of resources or decreasing a counter value, the operations do not take place.

Fix

Cast the `void*` pointer to the appropriate type. Perform the `delete` operation on the result of the cast.

For instance, if the `void*` pointer points to a `MyClass` object, cast the pointer to `MyClass*`.

Examples

Delete of `void*` Pointer

```
#include <iostream>

class MyClass {
public:
    explicit MyClass(int i):m_i(i) {}
    ~MyClass() {
        std::cout << "Delete MyClass(" << m_i << ")" << std::endl;
    }
private:
    int m_i;
};

void my_delete(void* ptr) {
    delete ptr;
}


int main() {
    MyClass* pt = new MyClass(0);
    my_delete(pt);
    return 0;
}
```

In this example, the function `my_delete` is designed to perform the `delete` operation on any type. However, in the function body, the `delete` operation acts on a `void*` pointer, `ptr`. Therefore, when you call `my_delete` with an argument of type `MyClass`, the `MyClass` destructor is not called.

Correction — Cast `void*` Pointer to `MyClass*`

One possible solution is to use a function template instead of a function for `my_delete`.

```
#include <iostream>

class MyClass {
public:
    explicit MyClass(int i):m_i(i) {}
    ~MyClass() {
        std::cout << "Delete MyClass(" << m_i << ")" << std::endl;
    }
private:
    int m_i;
};


template<typename T> void safe_delete(T*& ptr) {
    delete ptr;
    ptr = NULL;
}


int main() {
    MyClass* pt = new MyClass(0);
    safe_delete(pt);
    return 0;
}
```

Result Information

Group: Good practice
Language: C++
Default: Off
Command-Line Syntax: `DELETE_OF_VOID_PTR`
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Hard-coded buffer size

Size of memory buffer is a numerical value instead of symbolic constant

Description

This defect occurs when you use a numerical value instead of a symbolic constant when declaring a memory buffer such as an array.

Risk

Hard-coded buffer size causes the following issues:

- Hard-coded buffer size increases the likelihood of mistakes and therefore maintenance costs. If a policy change requires developers to change the buffer size, they must change every occurrence of the buffer size in the code.
- Hard-constant constants can be exposed to attack if the code is disclosed.

Fix

Use a symbolic name instead of a hard-coded constant for buffer size. Symbolic names include `const`-qualified variables, `enum` constants, or macros.

`enum` constants are recommended.

- Macros are replaced by their constant values after preprocessing. Therefore, they can expose the loop boundary.
- `enum` constants are known at compilation time. Therefore, compilers can optimize the loops more efficiently.

  `const`-qualified variables are usually known at run time.

Examples

Hard-Coded Buffer Size

```
int table[100];

void read(int);

void func(void) {
    for (int i=0; i<100; i++)
        read(table[i]);
}
```

In this example, the size of the array `table` is hard-coded.

Correction — Use Symbolic Name

One possible correction is to replace the hard-coded size with a symbolic name.

```
const int MAX_1 = 100;
#define MAX_2 100
```

```
enum { MAX_3 = 100 };

int table_1[MAX_1];
int table_2[MAX_2];
int table_3[MAX_3];

void read(int);

void func(void) {
    for (int i=0; i < MAX_1; i++)
        read(table_1[i]);
    for (int i=0; i < MAX_2; i++)
        read(table_2[i]);
    for (int i=0; i < MAX_3; i++)
        read(table_3[i]);
}
```

Result Information

Group: Good practice
Language: C | C++
Default: Off
Command-Line Syntax: HARD_CODED_BUFFER_SIZE
Impact: Low
CWE ID: 547

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Hard-coded loop boundary

Loop boundary is a numerical value instead of symbolic constant

Description

This defect occurs when you use a numerical value instead of symbolic constant for the boundary of a `for`, `while` or `do-while` loop.

Risk

Hard-coded loop boundary causes the following issues:

- Hard-coded loop boundary makes the code vulnerable to denial of service attacks when the loop involves time-consuming computation or resource allocation.
- Hard-coded loop boundary increases the likelihood of mistakes and maintenance costs. If a policy change requires developers to change the loop boundary, they must change every occurrence of the boundary in the code.

  For instance, the loop boundary is 10000 and represents the maximum number of client connections supported in a network server application. If the server supports more clients, you must change all instances of the loop boundary in your code. Even if the loop boundary occurs once, you have to search for a numerical value of `10000` in your code. The numerical value can occur in places other than the loop boundary. You must browse through those places before you find the loop boundary.

Fix

Use a symbolic name instead of a hard-coded constant for loop boundary. Symbolic names include `const`-qualified variables, `enum` constants or macros.`enum` constants are recommended because:

- Macros are replaced by their constant values after preprocessing. Therefore, they can expose the buffer size.
- `enum` constants are known at compilation time. Therefore, compilers can allocate storage for them more efficiently.

  `const`-qualified variables are usually known at run time.

Examples

Hard-Coded Loop Boundary

```
void performOperation(int);

void func(void) {
    for (int i=0; i<100; i++)
        performOperation(i);
}
```

In this example, the boundary of the `for` loop is hard-coded.

Correction — Use Symbolic Name

One possible correction is to replace the hard-coded loop boundary with a symbolic name.

```
const int MAX_1 = 100;
#define MAX_2 100
enum { MAX_3 = 100 };

void performOperation_1(int);
void performOperation_2(int);
void performOperation_3(int);

void func(void) {
    for (int i=0; i<MAX_1; i++)
        performOperation_1(i);
    for (int i=0; i<MAX_2; i++)
        performOperation_2(i);
    for (int i=0; i<MAX_3; i++)
        performOperation_3(i);
}
```

Result Information

Group: Good practice
Language: C | C++
Default: Off
Command-Line Syntax: `HARD_CODED_LOOP_BOUNDARY`
Impact: Low
CWE ID: 547

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Hard-coded object size used to manipulate memory

Memory manipulation with hard-coded size instead of `sizeof`

Description

This defect occurs on constants that are memory size arguments for memory functions such as `malloc` or `memset`.

Risk

If you hard code object size, your code is not portable to architectures with different type sizes. If the constant value is not the same as the object size, the buffer might or might not overflow.

Fix

For the size argument of memory functions, use `sizeof(`*`object`*`)`.

Examples

Assume 4-Byte Integer Pointers

```
#include <stddef.h>
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};
extern void fill_ints(int **matrix, size_t nb, size_t s);

void bug_hardcodedmemsize()
{
    size_t i, s;

    s = 4;
    int **matrix = (int **)calloc(SIZE20, s);
    if (matrix == NULL) {
        return; /* Indicate calloc() failure */
    }
    fill_ints(matrix, SIZE20, s);
    free(matrix);
}
```

In this example, the memory allocation function `calloc` is called with a memory size of 4. The memory is allocated for an integer pointer, which can be a more or less than 4 bytes depending on your target. If the integer pointer is not 4 bytes, your program can fail.

Correction — Use `sizeof(int *)`

When calling `calloc`, replace the hard-coded size with a call to `sizeof`. This change makes your code more portable.

```
#include <stddef.h>
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};
extern void fill_ints(int **matrix, size_t nb, size_t s);

void corrected_hardcodedmemsize()
{
    size_t i, s;

    s = sizeof(int *);
    int **matrix = (int **)calloc(SIZE20, s);
    if (matrix == NULL) {
        return; /* Indicate calloc() failure */
    }
    fill_ints(matrix, SIZE20, s);
    free(matrix);
}
```

Result Information

Group: Good Practice
Language: C | C++
Default: Off
Command-Line Syntax: `HARD_CODED_MEM_SIZE`
Impact: Low
CWE ID: 805

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2016b

Incorrect syntax of flexible array member size

Flexible array member defined with size zero or one

Description

This defect occurs when you do not use the standard C syntax to define a structure with a flexible array member.

Since C99, you can define a flexible array member with an unspecified size. For instance, `desc` is a flexible array member in this example:

```
struct record {
    size_t len;
    double desc[];
};
```

Prior to C99, you might have used compiler-specific methods to define flexible arrays. For instance, you used arrays of size one or zero:

```
struct record {
    size_t len;
    double desc[0];
};
```

This usage is not compliant with the C standards following C99.

Risk

If you define flexible array members by using size zero or one, your implementation is compiler-dependent. For compilers that do not recognize the syntax, an `int` array of size one has buffer for one `int` variable. If you try to write beyond this buffer, you can run into issues stemming from array access out of bounds.

If you use the standard C syntax to define a flexible array member, your implementation is portable across all compilers conforming with the standard.

Fix

To implement a flexible array member in a structure, define an array of unspecified size. The structure must have one member besides the array and the array must be the last member of the structure.

Examples

Flexible Array Member Defined with Size One

```
#include <stdlib.h>

struct flexArrayStruct {
  int num;
  int data[1];
};
```

```
unsigned int max_size = 100;

void func(unsigned int array_size) {
  if(array_size<= 0 || array_size > max_size)
      exit(1);
  /* Space is allocated for the struct */
  struct flexArrayStruct *structP
    = (struct flexArrayStruct *)
     malloc(sizeof(struct flexArrayStruct)
          + sizeof(int) * (array_size - 1));
  if (structP == NULL) {
    /* Handle malloc failure */
    exit(2);
  }

  structP->num = array_size;

  /*
   * Access data[] as if it had been allocated
   * as data[array_size].
   */
  for (unsigned int i = 0; i < array_size; ++i) {
    structP->data[i] = 1;
  }

  free(structP);
}
```

In this example, the flexible array member `data` is defined with a size value of one. Compilers that do not recognize this syntax treat `data` as a size-one array. The statement `structP->data[i] = 1;` can write to `data` beyond the first array member and cause out of bounds array issues.

Correction — Use Standard C Syntax to Define Flexible Array

Define flexible array members with unspecified size.

```
#include <stdlib.h>

struct flexArrayStruct{
  int num;
  int data[];
};

unsigned int max_size = 100;

void func(unsigned int array_size) {
  if(array_size<=0 || array_size > max_size)
      exit(1);

  /* Allocate space for structure */
  struct flexArrayStruct *structP
    = (struct flexArrayStruct *)
    malloc(sizeof(struct flexArrayStruct)
         + sizeof(int) * array_size);

  if (structP == NULL) {
    /* Handle malloc failure */
```

```
    exit(2);
  }

  structP->num = array_size;

  /*
   * Access data[] as if it had been allocated
   * as data[array_size].
   */
  for (unsigned int i = 0; i < array_size; ++i) {
    structP->data[i] = 1;
  }

  free(structP);
}
```

Result Information

Group: Good Practice
Language:C (checker disabled if the analysis runs on C90 code indicated by the option `-c-version c90`)
Default: Off
Command-Line Syntax: `FLEXIBLE_ARRAY_MEMBER_INCORRECT_SIZE`
Impact: Low

See Also

`Find defects (-checkers)|Hard-coded buffer size|Memory leak|Misuse of structure with flexible array member|Pointer access out of bounds|Unprotected dynamic memory allocation`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2018b

Incorrectly indented statement

Statement indentation incorrectly makes it appear as part of a block

Description

This defect occurs when the indentation of a statement makes it appear as part of an `if`, `else` or another block but the arrangement or lack of braces actually keeps the statement outside the block.

Risk

A developer or reviewer might incorrectly associate the statement with a block based on its indentation, leading to an incorrect assumption about the program logic.

For instance, in this example:

```
if(credentialsOK())
   login=1;
   setCookies();
```

the line `setCookies();` is not part of the `if` block, but the indentation suggests otherwise.

Fix

If you want a statement to be part of a block, make sure that the statement is within the braces associated with the block. To identify the extent of a block, on the **Source** pane, click the opening brace.

If an `if`, `else` or `while` statement has no braces following the condition, only the next line on an execution path upto a semicolon is considered part of the `if`, `else` or `while` block. If you want subsequent lines to be included in the block, wrap the lines in braces.

For instance, in the preceding example, to include both statements in the `if` block, use:

```
if(credentialsOK()) {
   login=1;
   setCookies();
}
```

Examples

`else` Statement Incorrectly Indented

```
int switch1, switch2;

void doSomething(void);
void doSomethingElse(void);

void func() {
    if(switch1)
        if(switch2)
            doSomething();
    else
```

```
        doSomethingElse();
}
```

In this example, the `else` is indented as if it is associated with the first `if`. However, the `else` is actually associated with the second `if`. The indentation does not match the actual association and might lead to incorrect assumptions about the program logic.

Correction – Use Braces Appropriately

If you want the `else` to be associated with the first `if`, use braces to mark the boundaries of the first `if` block.

```
int switch1, switch2;

void doSomething(void);
void doSomethingElse(void);

void func() {
    if(switch1) {
        if(switch2)
            doSomething();
    }
    else
        doSomethingElse();
}
```

Check Information

Group: Good practice
Language: C | C++
Default: Off
Command-Line Syntax: `INCORRECT_INDENTATION`
Impact: Low

See Also

`Find defects (-checkers)` | `Line with more than one statement` | `Semicolon on same line as if, for or while statement`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Large pass-by-value argument

Large argument passed by value between functions

Description

This defect occurs when a large input argument or return value is passed between functions by its value.

Risk

Copy by value creates a copy of the argument in the function body. If the argument is large, its copy uses up a substantial part of the stack space available to the function. The copy can also increase the execution time significantly.

Special considerations for return values: In C code, when a function returns by value, the return value is copied to the caller. Therefore, this defect appears on functions that have large return values. In C++ code, if a function return value is of class type, under certain conditions, the standard allows compilers to avoid copying the return value (C++98: Section 12.8, Item 15; C++11: Section 12.8, Item 31). Most compilers do not perform a copy in such cases. This behavior is called return value optimization. In such cases, Polyspace Bug Finder does not produce this defect if a large object is returned by value.

Fix

For variables larger than 64 bytes, pass the value by pointer or by reference. For structured variables, you can also refactor the variable type so that only some of the members are copied.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Large Function Argument

```
typedef struct s_userid {
    char name[2];
    int idnumber[100];
} userid;

char username(userid first) {
    return first.name[0];
}
```

The large structure, `userid`, is passed to the function `username`. Because `userid` is larger than 64 bytes, this function produces a large pass-by-value defect.

Correction — Pass By Reference

One possible correction is to pass the argument by reference instead of by value. In this corrected example, the pointer to a `userid` structure is passed instead of the actual structure.

```
typedef struct s_userid {
    char name[2];
```

```
    int idnumber[100];
} userid;

char username(userid *first) {
    return (*first).name[0];
}
```

Large Function Return Value

```
#include <stdlib.h>

#define initialSize 4
#define idSize 100

typedef struct {
    char initials[initialSize];
    int id[idSize];
} userId;

userId* getAddress(void);
assignValues(char*, int*);

userId username(void) {
    userId * newId = getAddress();
    assignValues((*newId).initials, (*newId).id);
    return *newId;
}
```

In this example, the function `username` returns a large structure `*newId` by value. When a function calls `username`, the value in `*newId` is copied to the caller.

Correction — Pass By Reference

One possible correction is to return the large structure by reference. In this corrected example, the pointer to structure `newId` is returned from the function `username`.

```
#include <stdlib.h>

#define initialSize 4
#define idSize 100

typedef struct {
    char initials[initialSize];
    int id[idSize];
} userId;

userId* getAddress(void);
assignValues(char*, int*);

userId * username(void) {
    userId * newId = getAddress();
    assignValues((*newId).initials, (*newId).id);
    return newId;
}
```

Check Information

Group: Good practice

Language: C | C++
Default: Off
Command-Line Syntax: PASS_BY_VALUE
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Macro terminated with a semicolon

Macro definition ends with a semicolon

Description

This defect occurs when a macro that is invoked at least once has a definition ending with a semicolon.

Risk

If a macro definition ends with a semicolon, the macro expansion can lead to unintended program logic in certain contexts, such as within an expression.

For instance, consider the macro:

```
#define INC_BY_ONE(x) ++x;
```

If used in the expression:

```
res = INC_BY_ONE(x)%2;
```

the expression resolves to:

```
res = ++x; %2;
```

The value of `x+1` is assigned to `res`, which is probably unintended. The leftover standalone statement `%2;` is valid C code and can only be detected by enabling strict compiler warnings.

Fix

Do not end macro definitions with a semicolon. Leave it up to users of the macro to add a semicolon after the macro when needed.

Alternatively, use inline functions in preference to function-like macros that involve statements ending with semicolon.

Examples

Spurious Semicolon in Macro Definition

```
#define WHILE_LOOP(n) while(n>0);

void performAction(int timeStep);

void main() {
    int loopIter = 100;
    WHILE_LOOP(loopIter) {
        performAction(loopIter);
        loopIter--;
    }
}
```

In this example, the defect occurs because the definition of the macro `WHILE_LOOP(n)` ends with a semicolon. As a result of the semicolon, the `while` loop has an empty body and the subsequent statements in the block run only once. It was probably intended that the loop must iterate 100 times.

Correction – Remove Semicolon from Macro Definition

Remove the trailing semicolon from the macro definition. Users of the macro can add a semicolon after the macro when needed. In this example, a semicolon is not required.

```
#define WHILE_LOOP(n) while(n>0)

void performAction(int timeStep);

void main() {
    int loopIter = 100;
    WHILE_LOOP(loopIter) {
        performAction(loopIter);
        loopIter--;
    }
}
```

Check Information

Group: Good practice
Language: C | C++
Default: Off
Command-Line Syntax: `SEMICOLON_TERMINATED_MACRO`
Impact: Low

See Also

`Find defects (-checkers)` | `Incorrectly indented statement` | `Macro with multiple statements` | `Semicolon on same line as if, for or while statement`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Line with more than one statement

Multiple statements on a line

Description

This defect occurs when, before preprocessing starts, the analysis detects additional text after the semicolon (`;`) on a line. A defect is not raised for comments, `for`-loop definitions, braces, or backslashes.

Risk

Use of one statement per line improves readability of the code. Since most statements in your code appear on a new line, use of multiple statements per line in a few cases within this arrangement can make code review difficult.

Fix

Write one statement per line.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Examples

Single-Line Initialization

```
int multi_init(void){
int abc = 4; int efg = 0; //defect

    return abc*efg;
}
```

In this example, `abc` and `efg` are initialized on the second line of the function as separate statements.

Correction — Comma-Separated Initialization

One possible correction is to use a comma instead of a semicolon to declare multiple variables on the same line.

```
int multi_init(void){
    int a = 4, b = 0;

    return a*b;
}
```

Correction — New Line for Each Initialization

One possible correction is to separate each initialization. By putting the initialization of `b` on the next line, the code longer raises a defect.

```
int multi_init(void){
    int a = 4;
    int b = 0;
```

```
    return a*b;
}
```

Single-Line Loops

```
int multi_loop(void){
    int a, b = 0;
    int index = 1;
    int tab[9] = {1,1,2,3,5,8,13,21};

    for(a=0; a < 3; a++) {b+=a;} // no defect

for(b=0; b < 3; b++) {a+=b; index=b;} //defect

while (index < 7) {index++; tab[index] = index * index;} //defect
    return a*b;
}
```

In this example, there are three loops coded on single lines, each with multiple semicolons.

- The first `for` loop has multiple semicolons. Polyspace does not raise a defect for multiple statements within a `for` loop declaration.
- Polyspace does raise a defect on the second `for` loop because there are multiple statements after the `for` loop declaration.
- The `while` loop also has multiple statements after the loop declaration. Polyspace raises a defect on this line.

Correction — New Line for Each Loop Statement

One possible correction is to use a new line for each statement after the loop declaration.

```
int multi_loop(void){
    int a, b = 0;
    int index = 1;
    int tab[9] = {1,1,2,3,5,8,13,21};

    for(a=0; a < 3; a++) {b+=a;}

    for(b=0; b < 3; b++){
      a+=b;
      index=b;
    }

    while (index < 7){
      index++;
      tab[index] = index * index;
    }
    return a*b;
}
```

Single-line Conditionals

```
int multi_if(void){

    int a, b = 1;
    if(a == 0) { a++;} // no defect
```

```
else if(b == 1) {b++; a *= b;} //defect
}
```

In this example, there are two conditional statements an: `if` and an `else if`. The `if` line does not raise a defect because only one statement follows the condition. The `else if` statement does raise a defect because two statements follow the condition.

Correction — New Lines for Multi-Statement Conditionals

One possible correction is to use a new line for conditions with multiple statements.

```
int multi_if(void){
    int a, b = 1;

    if(a == 0) a++;
    else if(b == 1){
      b++;
      a *= b;
    }
}
```

Check Information

Group: Good practice
Language: C | C++
Default: Off
Command-Line Syntax: `MORE_THAN_ONE_STATEMENT`
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2013b

Missing break of switch case

No comments at the end of switch case without a break statement

Description

This defect occurs when a switch case does not end in a `break` statement. If the case does not have a code comment after it, Polyspace assumes the missing break is not intentional and raises a defect.

Risk

Switch cases without break statements fall through to the next switch case. If this fall-through is not intended, the switch case can unintentionally execute code and end the switch with unexpected results.

Fix

If you do not want a break for the highlighted switch case, add a comment to your code to document why this case falls through to the next case. This comment removes the defect from your results and makes your code more maintainable.

If you forgot the break, add it before the end of the switch case.

Examples

Switch Without Break Statements

```
enum WidgetEnum { WE_W, WE_X, WE_Y, WE_Z } widget_type;

extern void demo_do_something_for_WE_W(void);
extern void demo_do_something_for_WE_X(void);
extern void demo_report_error(void);

void bug_missingswitchbreak(enum WidgetEnum wt)
{
    /*
      In this non-compliant code example, the case where widget_type is WE_W lacks a
      break statement. Consequently, statements that should be executed only when
      widget_type is WE_X are executed even when widget_type is WE_W.
    */
    switch (wt)
    {
      case WE_W:
        demo_do_something_for_WE_W();
      case WE_X:
        demo_do_something_for_WE_X();
      default:
        /* Handle error condition */
        demo_report_error();
    }
}
```

In this example, there are two cases without `break` statements. When `wt` is `WE_W`, the statements for `WE_W`, `WE_X`, and the `default` case execute because the program falls through the two cases without

a break. No defect is raised on the `default` case or last case because it does not need a break statement.

Correction — Add a Comment or break

To fix this example, either add a comment to mark and document the acceptable fall-through or add a break statement to avoid fall-through. In this example, case `WE_W` is supposed to fall through, so a comment is added to explicitly state this action. For the second case, a break statement is added to avoid falling through to the `default` case.

```
enum WidgetEnum { WE_W, WE_X, WE_Y, WE_Z } widget_type;

extern void demo_do_something_for_WE_W(void);
extern void demo_do_something_for_WE_X(void);
extern void demo_report_error(void);

void corrected_missingswitchbreak(enum WidgetEnum wt)
{
    switch (wt)
    {
      case WE_W:
        demo_do_something_for_WE_W();
        /* fall through to WE_X*/
      case WE_X:
        demo_do_something_for_WE_X();
        break;
      default:
        /* Handle error condition */
        demo_report_error();
    }
}
```

Result Information

Group: Good Practice
Language: C | C++
Default: Off
Command-Line Syntax: `MISSING_SWITCH_BREAK`
Impact: Low
CWE ID: 484

See Also

`Missing case for switch condition` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2016b

Missing overload of allocation or deallocation function

Only one function in an allocation-deallocation function pair is overloaded

Description

This defect occurs when you overload `operator new` but do not overload the corresponding `operator delete`, or vice versa.

Risk

You typically overload `operator new` to perform some bookkeeping in addition to allocating memory on the free store. Unless you overload the corresponding `operator delete`, it is likely that you omitted some corresponding bookkeeping when deallocating the memory.

The defect can also indicate a coding error. For instance, you overloaded the placement form of `operator new[]`:

```
void *operator new[](std::size_t count, void *ptr);
```

but the non-placement form of `operator delete[]`:

```
void operator delete[](void *ptr);
```

instead of the placement form:

```
void operator delete[](void *ptr, void *p );
```

Fix

When overloading `operator new`, make sure that you overload the corresponding `operator delete` in the same scope, and vice versa.

For instance, in a class, if you overload the placement form of `operator new`:

```
class MyClass {
   void* operator new  ( std::size_t count, void* ptr ){
   ...
   }
};
```

Make sure that you also overload the placement form of `operator delete`:

```
class MyClass {
   void operator delete  ( void* ptr, void* place ){
   ...
   }
};
```

To find the `operator delete` corresponding to an `operator new`, see the reference pages for `operator new` and `operator delete`.

Examples

Mismatch Between Overloaded `operator new` and `operator delete`

```
#include <new>
#include <cstdlib>

int global_store;

void update_bookkeeping(void *allocated_ptr, bool alloc) {
   if(alloc)
      global_store++;
   else
      global_store--;
}


void *operator new(std::size_t size, const std::nothrow_t& tag);
void *operator new(std::size_t size, const std::nothrow_t& tag) {
    void *ptr = (void*)malloc(size);
    if (ptr != nullptr)
        update_bookkeeping(ptr, true);
    return ptr;
}

void operator delete[](void *ptr, const std::nothrow_t& tag);
void operator delete[](void* ptr, const std::nothrow_t& tag) {
    update_bookkeeping(ptr, false);
    free(ptr);
}
```

In this example, the operators `operator new` and `operator delete[]` are overloaded but there are no overloads of the corresponding `operator delete` and `operator new[]` operators.

The overload of `operator new` calls a function `update_bookkeeping` to change the value of a global variable `global_store`. If the default `operator delete` is called, this global variable is unaffected, which might defy developer's expectations.

Correction — Overload the Correct Form of `operator delete`

If you want to overload `operator new`, overload the corresponding form of `operator delete` in the same scope.

```
#include <new>
#include <cstdlib>

int global_store;

void update_bookkeeping(void *allocated_ptr, bool alloc) {
   if(alloc)
      global_store++;
   else
      global_store--;
}
```

```
void *operator new(std::size_t size, const std::nothrow_t& tag);
void *operator new(std::size_t size, const std::nothrow_t& tag) {
    void *ptr = (void*)malloc(size);
    if (ptr != nullptr)
        update_bookkeeping(ptr, true);
    return ptr;
}

void operator delete(void *ptr, const std::nothrow_t& tag);
void operator delete(void* ptr, const std::nothrow_t& tag) {
    update_bookkeeping(ptr, false);
    free(ptr);
}
```

Result Information

Group: Good practice
Language: C++
Default: Off
Command-Line Syntax: `MISSING_OVERLOAD_NEW_DELETE_PAIR`
Impact: Low

See Also

`Find defects (-checkers)` | `Invalid deletion of pointer` | `Invalid free of pointer` | `Memory leak` | `Mismatched alloc/dealloc functions on Windows`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2019a

Missing reset of a freed pointer

Pointer `free` not followed by a reset statement to clear leftover data

Description

This defect occurs when a pointer is freed and not reassigned another value. After freeing a pointer, the memory data is still accessible. To clear this data, the pointer must also be set to NULL or another value.

Risk

Not resetting pointers can cause dangling pointers. Dangling pointers can cause:

- Freeing already freed memory.
- Reading from or writing to already freed memory.
- Hackers executing code stored in freed pointers or with vulnerable permissions.

Fix

After freeing a pointer, if it is not immediately assigned to another valid address, set the pointer to NULL.

Examples

Free Without Reset

```
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};

void missingfreedptrreset()
{
    static char *str = NULL;

    if (str == NULL)
        str = (char *)malloc(SIZE20);

    if (str != NULL)
        free(str);
}
```

In this example, the pointer `str` is freed at the end of the program. The next call to `bug_missingfreedptrrese` can fail because `str` is not NULL and the initialization to NULL can be invalid.

Correction — Redefine `free` to Free and Reset

One possible correction is to customize `free` so that when you free a pointer, it is automatically reset.

```
#include <stdlib.h>
enum {
```

```
    SIZE3   = 3,
    SIZE20  = 20
};

static void sanitize_free(void **p)
{
    if ((p != NULL) && (*p != NULL))
    {
        free(*p);
        *p = NULL;
    }
}

#define free(X) sanitize_free((void **)&X)

void missingfreedptrreset()
{
    static char *str = NULL;

    if (str == NULL)
        str = (char *)malloc(SIZE20);

    if (str != ((void *)0))
    {
        free(str);
    }
}
```

Result Information

Group: Good Practice
Language: C | C++
Default: Off
Command-Line Syntax: `MISSING_FREED_PTR_RESET`
Impact: Low
CWE ID: 415, 416, 825

See Also

`Use of previously freed pointer` | `Invalid free of pointer` | `Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2016b

Macro with multiple statements

Macro consists of multiple semicolon-terminated statements, enclosed in braces or not

Description

This defect occurs when a macro contains multiple semicolon-terminated statements, irrespective of whether the statements are enclosed in braces.

Risk

The macro expansion, in certain contexts such as an `if` condition or a loop, can lead to unintended program logic.

For instance, consider the macro:

```
#define RESET(x,y) \
   x=0; \
   y=0;
```

In an `if` statement such as:

```
if(checkSomeCondition)
   RESET(x,y);
```

the macro expands to:

```
if(checkSomething)
   x=0;
y=0;
```

which might be unexpected if you want both statements to be executed in an `if` block.

Fix

In a macro definition, wrap multiple statements in a `do...while(0)` loop.

For instance, in the preceding example, use the definition:

```
#define RESET(x,y) \
   do { \
     x=0; \
     y=0; \
   } while(0)
```

This macro is appropriate to expand in all contexts. The `while(0)` ensures that the statements are executed only once.

Alternatively, use inline functions in preference to function-like macros that involve multiple statements.

Note that the loop is required for the correct solution and wrapping the statements in braces alone does not fix the issue. The macro expansion can still lead to unintended code.

Examples

Macro with Multiple Statements

```
#define RESET(x,y) \
   x=0; \
   y=0;

void func(int *x, int *y, int resetFlag){
    if(resetFlag)
        RESET(x,y);
}
```

In this example, the defect occurs because the macro `RESET` consists of multiple statements.

Correction – Wrap Multiple Statements of Macro in `do-while` Loop

Wrap the statements of the macro in a `do..while(0)` loop in the macro definition.

```
#define RESET(x,y) \
   do { \
     x=0; \
     y=0; \
   } while(0)

void func(int *x, int *y, int resetFlag){
    if(resetFlag)
        RESET(x,y);
}
```

Check Information

Group: Good practice
Language: C | C++
Default: Off
Command-Line Syntax: `MULTI_STMT_MACRO`
Impact: Low

See Also

`Find defects (-checkers)` | `Incorrectly indented statement` | `Macro terminated with a semicolon` | `Semicolon on same line as if, for or while statement`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Possibly inappropriate data type for switch expression

`switch` expression has a data type other than char, short, int or enum

Description

This defect occurs when a `switch` expression has a data type other than `char`, `short`, `int` or `enum`.

The checker flags other integer data types such as boolean types, bit fields, or `long`.

Risk

It is preferred to use `char`, `short`, `int` or `enum` in `switch` expressions instead of:

- Boolean types, because a `switch` expression with a boolean type can be replaced with an `if` condition that evaluates the same expression. A `switch` expression is too heavy for a simple control flow based on a boolean condition.
- Bit field types, because bit field types imply memory restrictions. If you just want to specify a variable with a finite number of values, enumerations are preferred since they enable a more readable code.
- Types with size greater than `int` because a `switch` expression that requires a type with size greater than `int` implies too many case labels and can be possibly redesigned.

Non-integer types are not supported in `switch` expressions.

Fix

Use variables of `char`, `short`, `int` or `enum` data types in `switch` expressions.

Examples

Use of Inappropriate Types in `switch` Expressions

```
void func(bool s) {
    switch(s) {
        case 0: //Perform some operation
        break;
        case 1: //Perform another operation
        break;
    }
}
```

In this C++ example, the checker flags the use of a `bool` variable in a `switch` expression.

Correction – Use `if` Condition Instead of `switch`

If the `switch` expression indeed requires two values, use an `if` statement instead.

```
void func(bool s) {
    if(s) {
        //Perform some operation
```

```
        }
    else {
        //Perform another operation
        }
}
```

Correction – Use Different Data Type

If you anticipate adding more labels to the `switch` expression later, use a data type that can accommodate larger values.

```
void func(char s) {
    switch(s) {
        case 0: //Perform some operation
        break;
        case 1: //Perform another operation
        break;
        default: //Default behavior
    }
}
```

Check Information

Group: Good practice
Language: C | C++
Default: Off
Command-Line Syntax: `INAPPROPRIATE_TYPE_IN_SWITCH`
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Semicolon on same line as if, for or while statement

Semicolon on same line results in empty body of `if`, `for` or `while` statement

Description

This defect occurs when a semicolon on the same line as the last token of an `if`, `for` or `while` statement results in an empty body.

The checker makes an exception for the case where the `if` statement is immediately followed by an `else` statement:

```
if(condition);
else {
  ...
}
```

Risk

The semicolon following the if, for or while statement often indicates a programming error. The spurious semicolon changes the execution flow and leads to unintended results.

Fix

If you want an empty body for the `if`, `for` or `while` statement, wrap the semicolon in a block and place the block on a new line to explicitly indicate your intent:

```
if(condition)
   {;}
```

Otherwise, remove the spurious semicolon.

Examples

Spurious Semicolon

```
int credentialsOK(void);

void login () {
    int loggedIn = 0;
    if(credentialsOK());
      loggedIn = 1;
}
```

In this example, the spurious semicolon results in an empty `if` body. The assignment `loggedIn=1` is always performed. However, the assignment was probably to be performed only under a condition.

Correction – Remove Spurious Semicolon

If the semicolon was unintended, remove the semicolon.

```
int credentialsOK(void);
```

```
void login () {
    int loggedIn = 0;
    if(credentialsOK())
      loggedIn = 1;
}
```

Check Information

Group: Good practice
Language: C | C++
Default: Off
Command-Line Syntax: SEMICOLON_CTRL_STMT_SAME_LINE
Impact: Low

See Also

`Find defects (-checkers)|Incorrectly indented statement|Macro terminated with a semicolon|Macro with multiple statements`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Unmodified variable not const-qualified

Variable not `const`-qualified but variable value not modified during lifetime

Description

This defect occurs when a local variable is not `const`-qualified and one of the following statements is true during the variable lifetime:

- You do not perform write operations on the variable after initialization.
- When you perform write operations, you reassign the same constant value to the variable.

The checker considers a variable as modified if its address is assigned to a pointer or reference (unless it is a pointer or reference to a `const` variable), passed to another function, or otherwise used. In these situations, the checker does not suggest adding a `const` qualifier.

The checker flags arrays as candidates for `const`-qualification only if you do not perform write operations on the array elements at all after initialization.

Risk

`const`-qualifying a variable avoids unintended modification of the variable during later code maintenance. The `const` qualifier also indicates to a developer that the variable retains its initial value in the remainder of the code.

Fix

If you do not expect to modify a variable value during its lifetime, add the `const` qualifier to the variable declaration and initialize the variable at declaration.

If you expect the variable to be modified, see if the absence of a modification indicates a programming omission and fix the issue.

Examples

Missing `const` Qualification on Pointer

```
#include <string.h>

char returnNthCharacter (int n) {
    char* pwd = "aXeWdf10fg" ;
    char nthCharacter;

    for(int i=0; i < strlen(pwd); i++) {
        if(i==n)
            nthCharacter = pwd[i];
    }
    return nthCharacter;
}
```

In this example, the pointer `pwd` is not `const`-qualified. However, beyond initialization with a constant, it is not reassigned anywhere in the `returnNthCharacter` function.

Correction – Add `const` at Variable Declaration

If the variable is not intended to be modified, add the `const` qualifier at declaration. In this example, both the pointer and the pointed variable are not modified. Add a `const` qualifier to both the pointer and the pointed variable. Later modifications cannot reassign the pointer `pwd` to point at a different variable nor modify the value at the pointed location.

```
#include <string.h>

char returnNthCharacter (int n) {
    const char* const pwd = "aXeWdf10fg" ;
    char nthCharacter;

    for(int i=0; i < strlen(pwd); i++) {
        if(i==n)
            nthCharacter = pwd[i];
    }
    return nthCharacter;
}
```

Note that the checker only flags the missing `const` from the pointer declaration. The checker does not determine if the pointed location also merits a `const` qualifier.

Reassignment of Variable to Initial Value

```
void resetBuffer(int aCondition) {
    int addr = 0xff;
    if(aCondition){
        addr = 0xff;
    }
    else {
        addr = 0xff;
    }
}
```

In this example, the variable `addr` is initialized to a value and reassigned the same value twice. In larger code examples, such issues can easily arise from copy-paste errors.

Correction – Fix Programming Error

The reassignment in this example indicates a possible programming error. One possible correction is to fix the programming error and thereby avoid reassigning the same value.

```
void resetBuffer(int aCondition) {
    int addr = 0xff;
    if(aCondition){
        addr = 0x00;
    }
}
```

Check Information

Group: Good practice
Language: C | C++
Default: Off
Command-Line Syntax: `UNMODIFIED_VAR_NOT_CONST`
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

Unused parameter

Function prototype has parameters not read or written in function body

Description

This defect occurs when a function parameter is neither read nor written in the function body. The checker does not flag unused parameters in functions with empty bodies.

Risk

Unused parameters can indicate that the code is possibly incomplete. The parameter is possibly intended for an operation that you forgot to code.

If the copied objects are large, redundant copies can slow down performance.

Fix

Determine if you intend to use the parameters. Otherwise, remove parameters that you do not use in the function body.

You can intentionally have unused parameters. For instance, you have parameters that you intend to use later when you add enhancements to the function. Add a code comment indicating your intention for later use. The code comment helps you or a code reviewer understand why your function has unused parameters.

Alternatively, add a statement such as `(void)var;` in the function body. `var` is the unused parameter. You can define a macro that expands to this statement and add the macro to the function body.

Examples

Unused Parameter

```
void func(int* xptr, int* yptr, int flag) {
    if(flag==1) {
        *xptr=0;
    }
    else {
        *xptr=1;
    }
}

int main() {
    int x,y;
    func(&x,&y,1);
    return 0;
}
```

In this example, the parameter `yptr` is not used in the body of `func`.

Correction — Use Parameter

One possible correction is to check if you intended to use the parameter. Fix your code if you intended to use the parameter.

```
void func(int* xptr, int* yptr, int flag) {
    if(flag==1) {
        *xptr=0;
        *yptr=1;
    }
    else {
        *xptr=1;
        *yptr=0;
    }
}

int main() {
    int x,y;
    func(&x,&y,1);
    return 0;
}
```

Correction — Explicitly Indicate Unused Parameter

Another possible correction is to explicitly indicate that you are aware of the unused parameter.

```
#define UNUSED(x) (void)x

void func(int* xptr, int* yptr, int flag) {
    UNUSED(yptr);
    if(flag==1) {
        *xptr=0;
    }
    else {
        *xptr=1;
    }
}

int main() {
    int x,y;
    func(&x,&y,1);
    return 0;
}
```

Result Information

Group: Good practice
Language: C | C++
Default: Off
Command-Line Syntax: `UNUSED_PARAMETER`
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”

“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2015b

Use of a forbidden function

Function appears in a blacklist of forbidden functions

Description

This defect occurs when you use a function that appears in a blacklist of forbidden functions. To create the blacklist:

- List functions in an XML file in a specific syntax.

  Copy the template file `code-behavior-specifications-template.xml` from the folder *`polyspaceroot`*`\polyspace\verifier\cxx` to a writable location and modify the file. Enter each function in the file using the following syntax after existing similar entries:

  ```
  <function name="funcname" behavior="FORBIDDEN_FUNC"/>
  ```

  where *`funcname`* is the name of the function you want to blacklist.
- Specify this XML file as argument for the option `-code-behavior-specifications`.

Even if you enable this checker using the option `Find defects (-checkers)`, unless you specify a blacklist of functions, this checker stays disabled.

Risk

A function might be blacklisted for one of these reasons:

- The function can lead to many situations where the behavior is undefined leading to security vulnerabilities, and a more secure function exists.

  You can blacklist functions that are not explicitly checked by existing checkers such as `Use of dangerous standard function` or `Use of obsolete standard function`.
- The function is being deprecated as part of a migration, for instance, from C++98 to C++11.

  As part of a migration, you can make a list of functions that need to be replaced and use this checker to identify their use.

Fix

Replace the blacklisted function with an approved function.

When rolling out this checker to a group, project or organization, create a list of blacklist functions and their replacements so that results reviewers can consult the list and make appropriate replacements.

Examples

Use of Blacklisted Function

```
#include <csignal>
#include <iostream>
```

```
namespace
{
  volatile std::sig_atomic_t gSignalStatus;
}

void signal_handler(int signal)
{
  gSignalStatus = signal;
}

int main()
{
  // Install a signal handler
  std::signal(SIGINT, signal_handler);

  std::cout << "SignalValue: " << gSignalStatus << '\n';
  std::cout << "Sending signal " << SIGINT << '\n';
  std::raise(SIGINT);
  std::cout << "SignalValue: " << gSignalStatus << '\n';
}
```

Suppose you want to deprecate the `std::signal` function. Add the following to the template XML file after similar existing entries:

```
<function name="std::signal" behavior="FORBIDDEN_FUNC"/>
```

and specify the XML file with the option `-code-behavior-specifications`.

In the analysis results, all uses of the `std::signal` function are flagged by this checker.

Blacklisting C++ Overloaded Operators

```
class orderedPair {
        int var1;
        int var2;
    public:
        orderedPair() {
            var1 = 0;
            var2 = 0;
        }
        orderedPair(int arg1, int arg2) {
            var1 = arg1;
            var2 = arg2;
        }
        orderedPair& operator=(const orderedPair& rhs) {
            var1 = rhs.var1;
            var2 = rhs.var2;
            return *this;
        }
        orderedPair& operator+(orderedPair& rhs) {
            var1 += rhs.var1;
            var2 += rhs.var2;
            return *this;
        }
};

void main() {
```

```
    int one=1, zero=0, sum;
    orderedPair firstOrderedPair(one, one);
    orderedPair secondOrderedPair(zero, one);
    orderedPair sumPair;

    sum = zero + one;
    sumPair = firstOrderedPair + secondOrderedPair;
}
```

Suppose you want to identify all the locations where operator overloads in the `orderedPair` class are used. Add the overloaded operators to the template XML file:

```
    <function name="orderedPair::operator=" behavior="FORBIDDEN_FUNC"/>
    <function name="orderedPair::operator+" behavior="FORBIDDEN_FUNC"/>
```

and specify the XML file with the option `-code-behavior-specifications`.

The analysis identifies all calls to the overloaded operators and flags their use. Using this method, you can distinguish specific overloads of an operator instead of searching for and browsing through all instances of the operator.

Check Information

Group: Good practice
Language: C | C++
Default: Off
Command-Line Syntax: `FORBIDDEN_FUNC`
Impact: Low

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”
“Flag Deprecated or Unsafe Functions Using Bug Finder Checkers”

Introduced in R2020a

Use of setjmp/longjmp

`setjmp` and `longjmp` cause deviation from normal control flow

Description

This defect occurs when you use a combination of `setjmp` and `longjmp` or `sigsetjmp` and `siglongjmp` to deviate from normal control flow and perform non-local jumps in your code.

Risk

Using `setjmp` and `longjmp`, or `sigsetjmp` and `siglongjmp` has the following risks:

- Nonlocal jumps are vulnerable to attacks that exploit common errors such as buffer overflows. Attackers can redirect the control flow and potentially execute arbitrary code.
- Resources such as dynamically allocated memory and open files might not be closed, causing resource leaks.
- If you use `setjmp` and `longjmp` in combination with a signal handler, unexpected control flow can occur. POSIX does not specify whether `setjmp` saves the signal mask.
- Using `setjmp` and `longjmp` or `sigsetjmp` and `siglongjmp` makes your program difficult to understand and maintain.

Fix

Perform nonlocal jumps in your code using `setjmp/longjmp` or `sigsetjmp/siglongjmp` only in contexts where such jumps can be performed securely. Alternatively, use POSIX threads if possible.

In C++, to simulate throwing and catching exceptions, use standard idioms such as `throw` expressions and `catch` statements.

Examples

Use of `setjmp` and `longjmp`

```
#include <setjmp.h>
#include <signal.h>

extern int update(int);
extern void print_int(int);

static jmp_buf env;
void sighandler(int signum) {
    longjmp(env, signum);
}
void func_main(int i) {
    signal(SIGINT, sighandler);
    if (setjmp(env)==0) {
        while(1) {
            /* Main loop of program, iterates until SIGINT signal catch */
            i = update(i);
        }
    } else {
```

```
        /* Managing longjmp return */
        i = -update(i);
    }

    print_int(i);
    return;
}
```

In this example, the initial return value of `setjmp` is 0. The `update` function is called in an infinite `while` loop until the user interrupts it through a signal.

In the signal handling function, the `longjmp` statement causes a jump back to `main` and the return value of `setjmp` is now 1. Therefore, the `else` branch is executed.

Correction — Use Alternative to `setjmp` and `longjmp`

To emulate the same behavior more securely, use a `volatile` global variable instead of a combination of `setjmp` and `longjmp`.

```
#include <setjmp.h>
#include <signal.h>

extern int update(int);
extern void print_int(int);

volatile sig_atomic_t eflag = 0;

void sighandler(int signum) {
     eflag = signum;                   /* Fix: using global variable */
}

void func_main(int i) {
      /* Fix: Better design to avoid use of setjmp/longjmp */
    signal(SIGINT, sighandler);
    while(!eflag) {                   /* Fix: using global variable */
        /* Main loop of program, iterates until eflag is changed */
        i = update(i);
    }

    print_int(i);
    return;
}
```

Result Information

Group: Good practice
Language: C | C++
Default: Off
Command-Line Syntax: `SETJMP_LONGJMP_USE`
Impact: Low
CWE ID: 691

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”

“Address Polyspace Results Through Bug Fixes or Justifications”

External Websites
Linux man page for setjmp

Introduced in R2015b

Redundant expression in sizeof operand

`sizeof` operand contains expression that is not evaluated

Description

This defect occurs when a `sizeof` operand contains expressions whose evaluation does not affect the `sizeof` result. In place of the current expression in the `sizeof` operand, a data type, a variable or a simpler expression could have been used without any loss of functionality.

Risk

In situations flagged by this defect, the expression in the `sizeof` operand is needlessly complicated, reduces the code readability and adds to maintainability costs. The expression might also give a false impression about the result of the `sizeof` operand.

For instance, consider the expression:

```
sizeof(void (*[n])(int arr[U+V]))
```

The operand of `sizeof` is an array of `n` function pointers, each of type `void () (int*)`. The additional `U+V`, which is not evaluated, makes the full expression needlessly complicated. The expression also gives the false impression that the function pointer argument being an array of size `U+V` matters for the `sizeof` result.

Fix

The first event in the defect traceback shows where the redundant subexpression of the `sizeof` operand begins.

Simplify or completely remove the redundant expression. When possible, use a data type as the `sizeof` operand. For instance, in the preceding example, a simpler equivalent `sizeof` operation is:

```
sizeof(void (*[n])(int*))
```

If you want the expression to be evaluated, perform the evaluation in a separate statement.

Examples

Unnecessarily Complex Expression in `sizeof` Operand

```
void func() {
    int size1, size2, size3;
    char x = 0;
    short y = 0;
    int z = 0, w = 0;

    size1 = sizeof(x + y);
    size2 = sizeof(x + z);
    size3 = sizeof(z + w);
}
```

In this example, the defect checker flags the second and third `sizeof` operation because the expressions in the `sizeof` operand can be simplified without changing the `sizeof` result.

The checker does not flag the first operation because the expression in the `sizeof` operand cannot be simplified further without affecting the `sizeof` result.

Correction – Simplify Expression in `sizeof` Operand

Simplify the expression in the `sizeof` operand. In the following corrections, the `sizeof` results are the same as with the preceding expressions.

```
void func() {
    int size1, size2, size3;
    char x = 0;
    short y = 0;
    int z = 0, w = 0;

    size1 = sizeof(x + y);
    size2 = sizeof(z);
    size3 = sizeof(z);
}
```

Check Information

Group: Good practice
Language: C | C++
Default: Off
Command-Line Syntax: `SIZEOF_USELESS_OP`
Impact: Low

See Also

`Find defects (-checkers)` | `Side effect of expression ignored`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Justifications”

Introduced in R2020a

4

Functions, Properties, Classes, and Apps

polyspace-configure

(DOS/UNIX) Create Polyspace project from your build system at the DOS or UNIX command line

Syntax

```
polyspace-configure buildCommand

polyspace-configure [OPTIONS] buildCommand
```

Description

`polyspace-configure buildCommand` traces your build system and creates a Polyspace project with information gathered from your build system.

`polyspace-configure [OPTIONS] buildCommand` traces your build system and uses `-option value` to modify the default operation of `polyspace-configure`. Specify the modifiers before `buildCommand`, otherwise they are considered as options in the build command itself.

Examples

Create Polyspace Project from Makefile

This example shows how to create a Polyspace project if you use the command `make` *`targetName buildOptions`* to build your source code.

Create a Polyspace project specifying a unique project name. Use the `-B` or `-W` *`makefileName`* option with `make` so that the all prerequisite targets in the makefile are remade.

```
polyspace-configure  -prog myProject \
make -B targetName buildOptions
```

Open the Polyspace project in the Polyspace user interface.

Create Projects That Have Different Source Files from Same Build Trace

This example shows how to create different Polyspace projects from the same trace of your build system. You can specify which source files to include for each project.

Trace your build system without creating a Polyspace project by specifying the option `-no-project`. To ensure that all the prerequisite targets in your makefile are remade, use the appropriate `make` build command option, for instance `-B`.

```
polyspace-configure -no-project make -B
```

`polyspace-configure` stores the cache information and the build trace in default locations inside the current folder. To store the cache information and build trace in a different location, specify the options `-cache-path` and `-build-trace`.

Generate Polyspace projects by using the build trace information from the previous step. Specify a project name and use the `-include-sources` or `-exclude-sources` option to select which files to include for each project.

```
polyspace-configure -no-build -prog myProject \
-include-sources "glob_pattern"
```

`glob_pattern` is a glob pattern that corresponds to folders or files you filter in or out of your project. To ensure the shell does not expand the glob patterns you pass to `polyspace-configure`, enclose them in double quotes. For more information on the supported syntax for glob patterns, see "polyspace-configure Source Files Selection Syntax".

If you specified the options `-build-trace` and `-cache-path` in the previous step, specify them again.

Delete the trace file and cache folder.

```
rm -r polyspace_configure_cache polyspace_configure_built_trace
```

If you used the options `-build-trace` and `-cache-path`, use the paths and file names from those options.

Run Command-Line Polyspace Analysis from Makefile

This example shows how to run Polyspace analysis if you use the command `make` *`targetName buildOptions`* to build your source code. In this example, you use `polyspace-configure` to trace your build system but do not create a Polyspace project. Instead you create an options file that you can use to run Polyspace analysis from command-line.

Create a Polyspace options file specifying the `-output-options-file` command. Use the `-B` or `-W` *`makefileName`* option with `make` so that all prerequisite targets in the makefile are remade.

```
polyspace-configure -output-options-file\
 myOptions make -B targetName buildOptions
```

Use the options file that you created to run a Polyspace analysis at the command line:

```
polyspace-bug-finder -options-file myOptions
```

Input Arguments

`buildCommand` — Command for building source code

build command

Build command specified exactly as you use to build your source code.

Example: `make -B`, `make -W` *`makefileName`*

`[OPTIONS]` — Options for changing default operation of `polyspace-configure`

single option starting with `-`, followed by argument | multiple space-separated option-argument pairs

Basic Options

Option	Argument	Description
`-prog`	Project name	Project name that appears in the Polyspace user interface. The default is `polyspace`. If you do not use the option `-output-project`, the `-prog` argument also sets the project name. **Example:** `-prog myProject` creates a project that has the name `myProject` in the user interface. If you do not use the option `-output-project`, the project name is also `myProject.psrprj`.
`-author`	Author name	Name of project author. **Example:** `-author jsmith`
`-output-project`	Path	Project file name and location for saving project. The default is the file `polyspace.psprj` in the current folder. **Example:** `-output-project ../myProjects/project1` creates a project `project1.psprj` in the folder with the relative path `../myProjects/`.
`-output-options-file`	File name	Option to create a Polyspace analysis options file. Use this file for command-line analysis using `polyspace-bug-finder`.

`-allow-build-error`	None	Option to create a Polyspace project even if an error occurs in the build process.
		
		If an error occurs, the build trace log shows the following message:
		
		`polyspace-configure (polyspaceConfigure)`
		`  ERROR: build command`
		`  *command_name* fail [status=*status_value*]`
		
		`command_name` is the build command name that you use and *`status_value`* is the non-zero exit status or error level that indicates which error occurred in your build process.

`-allow-overwrite`	None	Option to overwrite a project with the same name, if it exists. By default, `polyspace-configure (polyspaceConfigure)` throws an error if a project with the same name already exists in the output folder. Use this option to overwrite the project.
`-silent` (default) `-verbose`	None	Option to suppress or display additional messages from running `polyspace-configure (polyspaceConfigure)`.
`-help`	None	Option to display the full list of `polyspace-configure (polyspaceConfigure)` commands

Option	Argument	Description
`-debug`	None	Option to store debug information for use by MathWorks technical support. This option has been superseded by the option `-easy-debug`.
`-easy-debug`	Path	Option to store debug information for use by MathWorks technical support. After a `polyspace-configure (polyspaceConfigure)` run, the path provided contains a zipped file ending with `pscfg-output.zip`. If the run fails to create a complete Polyspace project or options file, send this zipped file to MathWorks Technical Support for further debugging. The zipped file does not contain source files traced in the build. See also “Errors in Project Creation from Build Systems”.

Options to Create Multiple Modules

Option	Argument	Description
`-module`	None	Option to create a separate options file for each binary created in build system. You can only create separate options files for different binaries. You cannot create multiple modules in a Polyspace project (for running in the Polyspace user interface). Use this option only for build systems that use GNU and Visual C++ compilers. See also “Modularize Polyspace Analysis by Using Build Command”.
`-output-options-path`	Path name	Location where generated options files are saved. Use this option together with the option `-module`. The options files are named after the binaries created in the build system.

Advanced Options

Option	Argument	Description
`-compiler-config`	Path and file name	Location and name of compiler configuration file. The file must be in a specific format. For guidance, see the existing configuration files in *polyspaceroot*`\polyspace\configure\compiler_configuration\`. For information on the contents of the file, see “Compiler Not Supported for Project Creation from Build Systems”. **Example:** `-compiler-configuration myCompiler.xml`
`-no-project`	None	Option to trace your build system without creating a Polyspace project and save the build trace information. Use this option to save your build trace information for a later run of `polyspace-configure (polyspaceConfigure)` with the `-no-build` option.
`-no-build`	None	Option to create a Polyspace project using previously saved build trace information. To use this option, you must have the build trace information saved from an earlier run of `polyspace-configure (polyspaceConfigure)` with the `-no-project` option. If you use this option, you do not need to specify the `buildCommand` argument.

Option	Argument	Description

`-no-sources`	None	Option to create a Polyspace options file that does not contain the source file specifications.
		
		Use this option when you intend to specify the source files by other means. For instance, you can use this option when:
		
		• Running Polyspace on AUTOSAR-specific code.
		
		You want to create an options file that traces your build command for the compiler options:
		
		`-output-options-file options.txt -no-sources`
		
		You later append this options file when extracting source file names from ARXML specifications and running the subsequent Code Prover analysis with `polyspace-autosar`
		
		`-extra-options-file options.txt`
		
		See also “Run Polyspace on AUTOSAR Code Using Build Command” (Polyspace Code Prover).
		• Running Polyspace in Eclipse™.
		
		Your source files are already specified in your Eclipse project. When running a Polyspace analysis, you want to specify an options file that has the compilation options only.

Option	Argument	Description
`-extra-project-options`	Options to use for subsequent Polyspace analysis. For instance, `"-stubbed-pointers-are-unsafe"`.	Options that are used for subsequent Polyspace analysis. Once a Polyspace project is created, you can change some of the default options in the project. Alternatively, you can pass these options when tracing your build command. The flag `-extra-project-options` allows you to pass additional options. Specify multiple options in a space separated list, for instance `"-allow-negative-operand-in-shift -stubbed-pointers-are-unsafe"`. Suppose you have to set the option `-stubbed-pointers-are-unsafe` for every Polyspace project created. Instead of opening each project and setting the option, you can use this flag when creating the Polyspace project: `-extra-project-options "-stubbed-pointers-are-unsafe"` For the list of options available, see: • “Analysis Options” • • If you are creating an options file instead of a Polyspace project from your build command, do not use this flag.
`-tmp-path`	Path	Location of folder where temporary files are stored.
`-build-trace`	Path and file name	Location and name of file where build information is stored. The default is `./polyspace_configure_build_trace.log`. **Example:** `-build-trace ../build_info/trace.log`

`-include-sources`
 	
`-exclude-sources`
	Glob pattern	Option to specify which source files `polyspace-configure (polyspaceConfigure)` includes in, or excludes from, the generated project. You can combine both options together.
		
		A source file is included if the file path matches the glob pattern that you pass to `-include-sources`.
		
		A source file is excluded if the file path matches the glob pattern that you pass to `-exclude-sources`.

Option	Argument	Description

`-print-included-sources`
 	
`-print-excluded-sources`
	None	Option to print the list of source files that `polyspace-configure (polyspaceConfigure)` includes in, or excludes from, the generated project. You can combine both options together. The output displays the full path of each file on a separate line.
		
		Use this option to troubleshoot the glob patterns that you pass to `-include-sources` or `-exclude-sources`. You can see which files match the pattern that you pass to `-include-sources` or `-exclude-sources`.

Cache Control Options

These options are primarily useful for debugging. Use the options if `polyspace-configure (polyspaceConfigure)` fails and MathWorks Technical Support asks you to use the option and provide the cached files. Starting R2020a, the option `-easy-debug` provides an easier way to provide debug information. See “Contact Technical Support About Issues with Running Polyspace”.

Option	Argument	Description

`-no-cache`
 	
`-cache-sources` (default)
 	
`-cache-all-text`
 	
`-cache-all-files`
	None	Option to perform one of the following:
		
		• `-no-cache`: Not create a cache
		• `-cache-sources`: Cache text files temporarily created during build for later use by `polyspace-configure (polyspaceConfigure)`.
		• `-cache-all-text`: Cache all text files including sources and headers.
		• `-cache-all-files`: Cache all files including binaries.
		
		Typically, you cache temporary files created by your build command to debug issues in tracing the command.

`-cache-path`	Path	Location of folder where cache information is stored. **Example:** `-cache-path ../cache`
`-keep-cache` `-no-keep-cache` (default)	None	Option to preserve or clean up cache information after `polyspace-configure (polyspaceConfigure)` completes execution. If `polyspace-configure (polyspaceConfigure)` fails, you can provide this cache information to technical support for debugging purposes.

See Also

Topics
“Requirements for Project Creation from Build Systems”
“Compiler Not Supported for Project Creation from Build Systems”
“Modularize Polyspace Analysis by Using Build Command”

Introduced in R2013b

polyspace-bug-finder Command

(DOS/UNIX) Run a Bug Finder analysis from the DOS or UNIX command line

Syntax

```
polyspace-bug-finder
polyspace-bug-finder -sources sourceFiles [OPTIONS]

polyspace-bug-finder -sources-list-file listOfSources [OPTIONS]

polyspace-bug-finder -options-file optFile

polyspace-bug-finder -h[elp]
```

Description

`polyspace-bug-finder [OPTIONS]` runs a Bug Finder analysis if your current folder contains a `sources` subfolder with source files (`.c` or `.cxx` files). The analysis considers files in `sources` and all subfolders under `sources`.

`polyspace-bug-finder -sources sourceFiles [OPTIONS]` runs a Bug Finder analysis on the source file(s) `sourceFiles`. You can customize the analysis with additional options.

`polyspace-bug-finder -sources-list-file listOfSources [OPTIONS]` runs a Bug Finder analysis on the source files listed in the text file `listOfSources`. You can customize the analysis with additional options. Using a sources list file is recommended when you have many source files. By keeping the list of sources in a text file, the command is shorter and updates to the list are easier.

`polyspace-bug-finder -options-file optFile` runs a Bug Finder analysis with the options specified in the option file. When you have many analysis options, an options file makes it easier to run the same analysis again.

`polyspace-bug-finder -h[elp]` lists a summary of possible analysis options.

Examples

Run Analysis by Directly Specifying Options

Run a local Bug Finder analysis by specifying analysis options in the command itself. This example uses source files from a demo Polyspace Bug Finder example. To run this example, replace *`polyspaceroot`* with the path to your Polyspace installation, for example `C:\Program Files \Polyspace\R2019a`.

Run an analysis on `numerical.c` and `programming.c`, checking for MISRA C:2012 mandatory rules, programming and numerical defects, and using GNU 4.7 compiler settings. This example command is split by ^ characters for readability. In practice, you can put all commands on one line.

```
polyspaceroot\polyspace\bin\polyspace-bug-finder^
 -sources ^
```

```
polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources\numerical.c,^
polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources\programming.c ^
-compiler gnu4.7 -misra3 mandatory -checkers numerical,programming ^
-author jlittle -prog myProject -results-dir C:\Polyspace_Workspace\Results\
```

Open the results.

```
polyspaceroot\polyspace\bin\polyspace C:\Polyspace_Workspace\Results\^
ps_results.psbf
```

To rerun the analysis, you must rerun it from the command line.

Run Local Analysis with Options File

Run a local Bug Finder analysis by specifying analysis options with an options. This example uses source files from a demo Polyspace Bug Finder example. To run this example, replace *polyspaceroot* with the path to your Polyspace installation, for example `C:\Program Files \Polyspace\R2019a`.

Save this text to a text file called `myOptionsFile.txt`.

```
# Options for analyzing numerical.c and programming.c
-sources polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources\numerical.c
-sources polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources\programming.c
-compiler gnu4.7
-misra3 mandatory
-checkers numerical,programming
-author jlittle
-prog myProject
-results-dir C:\Polyspace_Workspace\Results\
```

Run the analysis with the options specified in the text file.

```
polyspaceroot\polyspace\bin\polyspace-bug-finder -options-file myOptionsFile.txt
```

Open the results.

```
polyspaceroot\polyspace\bin\polyspace C:\Polyspace_Workspace\Results\^
ps_results.psbf
```

To rerun the analysis, you must rerun it from the command line.

Input Arguments

`sourceFiles` — Comma-separated names of C or C++ files to analyze

`-sources` string

Comma-separated C or C++ source file names, specified as `-sources` followed by a string. If the files are not in the current folder (`pwd`), `sourceFiles` must include a full or relative path. To avoid errors because of paths with spaces, add quotes `" "` around the path. For more information, see `-sources`.

If your current folder contains a `sources` subfolder with the source files, you can omit the `-sources` flag. The analysis considers files in `sources` and all subfolders under `sources`.

Example: `-sources myFile.c`, `-sources C:\mySources\myFile1.c,C:\mySources\myFile2.c`

`listOfSources` — Text file listing names of C or C++ files to analyze
`-sources-list-file` file

Text file which lists the name of C or C++ files, specified as `-sources-list-file` followed by the file. If the files are not in the current folder (`pwd`), `listOfSources` must include a full or relative path. To avoid errors because of paths with spaces, add quotes `" "` around the path. For more information, see `-sources-list-file`.

Example: `-sources-list-file filename.txt`, `-sources-list-file "C:\ps_analysis\source_files.txt"`

`[OPTIONS]` — Analysis option and corresponding value
option syntax

Analysis options and their corresponding values, specified by the option name and if applicable value. For syntax specifications, see the individual analysis option reference pages.

Example: `-lang C-CPP -compiler diab`

`optFile` — Text file listing analysis options and values
`-options-file` file

Text file listing analysis options and values, specified as `-options-file` followed by the file. For more information, see `-options-file`.

Example: `-options-file opts.txt`, `-options-file "C:\ps_analysis\options.txt"`

See Also

`polyspaceBugFinder`

Topics
“Run Polyspace Analysis from Command Line”
“Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”
“Analysis Options”

Introduced in R2013b

polyspace-report-generator

(DOS/UNIX) Generate reports for Polyspace analysis results stored locally or on Polyspace Access

Syntax

```
polyspace-report-generator -template <template> [OPTIONS]
polyspace-report-generator -generate-results-list-file [-results-dir <
FOLDER>] [-set-language-english]
polyspace-report-generator -generate-variable-access-file [-results-dir <
FOLDER>] [-set-language-english]

polyspace-report-generator -template <template> -host <HOSTNAME> -run-id <
RUN_ID> [ACCESS_OPTIONS] [OPTIONS]
polyspace-report-generator -generate-results-list-file -host <HOSTNAME> -run-
id <RUN_ID> [ACCESS_OPTIONS] [-set-language-english]
polyspace-report-generator -generate-variable-access-file -host <HOSTNAME> -
run-id <RUN_ID> [ACCESS_OPTIONS] [-set-language-english]
polyspace-report-generator -configure-keystore
```

Description

`polyspace-report-generator -template <template> [OPTIONS]` generates a report by using `TEMPLATE` for the local analysis results that you specify with `OPTIONS`.

By default, reports for results from `project-name` are stored as `project-name_report-name` in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the results folder of `project-name`.

`polyspace-report-generator -generate-results-list-file [-results-dir < FOLDER>] [-set-language-english]` exports the analysis results stored locally in `FOLDER` to a tab-delimited text file. The file contains the result information available on the **Results List** pane in the user interface. For more information on the exported results list, see “View Exported Results”.

By default, the results file for results from `project-name` is stored in the `PathToFolder \Polyspace-Doc` folder. `PathToFolder` is the results folder of `project-name`.

`polyspace-report-generator -generate-variable-access-file [-results-dir < FOLDER>] [-set-language-english]` exports the list of global variables in your code from the Code Prover analysis stored locally in `FOLDER` to a tab-delimited text file. The file contains the information available on the **Variable Access** pane in the user interface. For more information on the exported variables list, see “View Exported Variable List” (Polyspace Code Prover).

By default, the variables file for results from `project-name` is stored in the `PathToFolder \Polyspace-Doc` folder. `PathToFolder` is the results folder of `project-name`.

`polyspace-report-generator -template <template> -host <HOSTNAME> -run-id < RUN_ID> [ACCESS_OPTIONS] [OPTIONS]` generates a report by using `TEMPLATE` for the analysis results run `RUN_ID` stored on Polyspace Access. `HOSTNAME` is the fully qualified host name of the machine that hosts Polyspace Access.

By default, reports for results from `project-name` are stored as `project-name_report-name` in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the path from which you call the command.

`polyspace-report-generator -generate-results-list-file -host <HOSTNAME> -run-id <RUN_ID> [ACCESS_OPTIONS] [-set-language-english]` exports the analysis results run `RUN_ID` stored on Polyspace Access to a tab-delimited text file. The file contains the result information available on the **Results List** pane in the Polyspace Access web interface. `HOSTNAME` is the fully qualified host name of the machine that hosts Polyspace Access. For more information on the exported results list, see “Results List” (Polyspace Bug Finder Access).

By default, the results file for results from `project-name` is stored in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the path from which you call the command.

`polyspace-report-generator -generate-variable-access-file -host <HOSTNAME> -run-id <RUN_ID> [ACCESS_OPTIONS] [-set-language-english]` exports the list of global variables in your code from the Code Prover analysis run `RUN_ID` stored on Polyspace Access to a tab-delimited text file. The file contains the information available on the **Variable Access** pane in the Polyspace Access web interface. `HOSTNAME` is the fully qualified host name of the machine that hosts Polyspace Access. For more information on the exported variables list, see “View Exported Variable List” (Polyspace Code Prover).

By default, the variables file for results from `project-name` is stored in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the path from which you call the command.

`polyspace-report-generator -configure-keystore` configures the report generator to communicate with Polyspace Access over HTTPS.

Run this one-time configuration step if Polyspace Access is configured to use the HTTPS protocol and you do not have a Polyspace Bug Finder desktop license, or you have a desktop license but you have not configured the desktop UI to communicate with Polyspace Access over HTTPS. Before running this command, generate a client keystore to store the SSL certificate that Polyspace Access uses for HTTPS. See “Generate a Client Keystore” (Polyspace Bug Finder Access).

Examples

Generate PDF Reports for Analysis Results Stored Locally

You can generate multiple reports for analysis results that you store locally.

Create a variable `template_path` to store the path to the report templates and create a variable `report_templates` to store a comma-separated list of templates to use.

```
SET template_path="C:\Program Files"\Polyspace\R2019a\toolbox\polyspace^
\psrptgen\templates\bug_finder
SET report_templates=%template_path%\BugFinder.rpt,^
%template_path%\CodingStandards.rpt
```

Generate the reports from the templates that you specified in `report_templates` for analysis results of Polyspace project `myProject`.

```
 polyspace-report-generator -template %report_templates% ^
-results-dir C:\Polyspace_Workspace\myProject\Module_1\BF_Result ^
-format PDF
```

The command generates two PDF reports, `myProject_BugFinder.PDF` and `myProject_CodingStandards.PDF`. The reports are stored in `C:\Polyspace_Workspace\myProject\Module_1\BF_Result\Polyspace-Doc`. For more information on the content of the reports, see `Bug Finder and Code Prover report (-report-template)`.

Configure Report Generator with Client Keystore

If you configure Polyspace Access to use the HTTPS protocol, you must generate a client keystore where you store the SSL certificate that Polyspace Access uses, and configure `polyspace-report-generator` to use that keystore. See “Generate a Client Keystore” (Polyspace Bug Finder Access). This one-time configuration enables the report generator to communicate with Polyspace Access over HTTPS.

To configure the report generator with a client keystore, use the `polyspace-report-generator -configure-keystore` command. Follow the prompts to provide the URL you use to log into Polyspace Access, the full path to the keystore file you generated, and the keystore password.

```
polyspace-report-generator -configure-keystore
Location: US, user name: jsmit, id: 62600@us-jsmith, print mode: false
Enter the Polyspace Access URL using form  http[s]://<host>:<port> :
https://myAccessServer:9443
Enter full path to client keystore file :
C:\R2019b\ssl\client-cert.jks
Enter client keystore password :

The keystore has been configured
```

You must run the keystore configuration command again if:

- The Polyspace Access URL changes, for instance if you use a different port number.
- The path to the keystore file changes.
- The keystore password changes.

Generate Report and Variables List from Polyspace Access

Note To use the command-line for generating reports of results stored on Polyspace Access, you must have a Polyspace Bug Finder Server or Polyspace Code Prover Server installation.

Suppose that you want to generate a report and export the variables list for the results of a Code Prover analysis stored on the Polyspace Access database.

To connect to Polyspace Access, provide a host name and your login credentials including your encrypted password. To encrypt your password, use the `polyspace-access` command and enter your user name and password at the prompt.

```
polyspace-access -encrypt-password
login: jsmith
password:
CRYPTED_PASSWORD LAMMMEACDMKEFELKMNDCONEAPECEEKPL
Command Completed
```

Store your Polyspace Access login credentials in a variable `LOGIN`.

```
set LOGIN=-host jsmith ^
-encrypted-password LAMMMEACDMKEFELKMNDCONEAPECEEKPL
```

To specify project results on the Polyspace Access, specify the run ID of the project. To obtain a list of projects with their latest run ID, use the `polyspace-access` with option `-list-project`.

```
polyspace-access -host myAccessServer %LOGIN% -list-project
Connecting to https://myAccessServer:9443
Connecting as jsmith
Get project list with the last Run Id
Restricted/Code_Prover_Example (Code Prover) RUN_ID 14
public/Bug_Finder_Example (Bug Finder) RUN_ID 24
public/CP/Code_Prover_Example (Polyspace Code Prover) RUN_ID 16
public/Polyspace (Code Prover) RUN_ID 28
Command Completed
```

For more information on the command, see `polyspace-access`.

Generate a `Developer` report for results with run ID `16` from the Polyspace Access instance with host name `myAccessServer`. The URL of this instance of Polyspace Access is `https://myAccessServer:9443`.

```
SET template_path=^
"C:\Program Files\Polyspace\R2019a\toolbox\polyspace\psrptgen\templates"

polyspace-report-generator %LOGIN% ^
-template %template_path%\Developer.rpt ^
-host myAccessServer ^
-run-id 16 ^
-output-name myReport
```

The command creates report `myReport.docx` by using the template that you specify. The report is stored in folder `Polyspace-Doc` on the path from which you called the command.

Generate a tab-delimited text file that contains a list of global variables in your code for the specified analysis results.

```
polyspace-report-generator %LOGIN%^
-generate-variable-access-file ^
-host myAccessServer ^
-run-id 16
```

The list of global variables `Variable_View.txt` is stored in the same folder as the generated report. For more information on the exported variables list, see “View Exported Variable List” (Polyspace Code Prover).

Input Arguments

`template` — path to report template file

string

Path to the report template that you use to generate an analysis report. To generate multiple reports, specify a comma-separated list of report template paths (do not put a space after the commas). The templates are available in *`polyspaceroot`*`\toolbox\polyspace\psrptgen\templates\` as `.rpt` files. Here, *`polyspaceroot`* is the Polyspace installation folder. For more information on the available templates, see `Bug Finder and Code Prover report (-report-template)`.

This option is not compatible with `-generate-variable-access-file` and `-generate-results-list-file`.

Example: `C:\Program Files\Polyspace\R2019a\toolbox\polyspace\psrptgen\templates\Developer.rpt`

Example: `TEMPLATE_PATH\BugFinder.rpt,TEMPLATE_PATH\CodingStandards.rpt`

`FOLDER` — Analysis results folder path
string

Path to the folder containing analysis results for which you generate a report, or analysis results from which you export a list of results or global variables (Code Prover). To generate a report for multiple verifications, specify a comma-separated list of folder paths (do not put a space after the commas). If you do not specify a folder path, the command generates a report for analysis results in the current folder.

Example: `C:\Polyspace_Workspace\My_project\Module_1\results`

Example: `C:\Polyspace_Workspace\My_project\Module_2\results,C:\Polyspace_Workspace\My_project\Module_3\other_results`

`HOSTNAME` — Polyspace Access machine host name
string

Fully qualified host name of the machine that hosts the Polyspace Access **Gateway API** service. You must specify a host name to generate a report for results on the Polyspace Access database.

Example: `my-company-server`

`RUN_ID` — Polyspace Access run ID
integer

Run ID of the project findings for which you generate a report. Polyspace assigns a unique run ID to each analysis run that you upload to the Polyspace Access. To get the run ID of project findings, use the command `polyspace-access` with option `-list-project`.

Example: 4

`OPTIONS` — Options for generated report
string

Option	Description
`-format` *`HTML`* | *`PDF`* | *`WORD`*	File format of the report that you generate. By default, the command generates a WORD document. To generate reports in multiple formats, specify a comma-separated list of formats. (Do not put a space after the commas). For instance, `-format PDF,HTML`. This option is not compatible with `-generate-variable-access-file` and `-generate-results-list-file`.

Option	Description
`-output-name` *`outputName`*	Name of the generated report or folder name if you generate multiple reports. The command stores *`outputName`* on the path from which you call the command. To store the generated files in a different folder, specify the full path of the folder, for instance `-output-name C:\PathTo\OtherFolder`.
`-results-dir` *`FOLDER_1,...,FOLDER_N`*	Path to the locally stored results folder. To generate reports for multiple analyses, specify a comma-separated list of folder path. (Do not put a space after the commas). For example: `-results-dir folderPath1,folderPath2`
`-set-language-english`	Generate the report in English. Use this option if your display language is set to another language.
`-h`	Display the help information.

`ACCESS_OPTIONS` — Options for Polyspace Access
string

Option	Description
`-host` *`HOST_NAME`*	Fully qualified host name of the machine that hosts the Polyspace Access **Gateway API** service. This option is mandatory when you generate reports for results stored on the Polyspace Access database.
`-run-id` *`RUN_ID`*	Run ID of the project. Polyspace assigns a unique run ID to each analysis run that you upload. To get the last run ID of a project, use the `-list-project` option of the `polyspace-access` command. For more information on the command, see `polyspace-access`. This option is mandatory when you generate reports for results stored on the Polyspace Access database.
`-all-units`	Specify this option to generate a report for all units from a `unit by unit` analysis. When you use this option, specify the run ID of only one unit with `-run-id`. The command includes the other units from the analysis in the report.
`-port` *`portNumber`*	Port number of the Polyspace Access instance. Default value is `9443`.

Option	Description
`-protocol` *`http`* `|` *`https`*	HTTP protocol used to connect to Polyspace Access. Default value is `https`.
`-login` *`username`* `-encryted-password` *`ENCRYPTED_PASSWD`*	Credentials that you use to log into Polyspace Access. The argument of `-encrypted-password` is the output of the `polyspace-access -encrypt-password` command. For more information on the command, see `polyspace-access`.

See Also

Introduced in R2013b

polyspace-results-repository

(DOS/UNIX) Upload, download and otherwise interact with results in the Polyspace Metrics repository

Syntax

```
polyspace-results-repository -upload resultsFolder -product productName -prog
projectName -verif-version versionNumber [OPTIONS]

polyspace-results-repository -download resultsFolder -product productName -
prog projectName -verif-version versionNumber [OPTIONS]

polyspace-results-repository -get-projects-list -product productName
polyspace-results-repository -get-versions-list -product productName -prog
projectName
polyspace-results-repository -get-run-numbers-list -product productName -prog
projectName -verif-version versionNumber
polyspace-results-repository -get-files-list -product productName -prog
projectName -verif-version versionNumber [OPTIONS]

polyspace-results-repository -get-sqo-id -product productName -prog
projectName -verif-version versionNumber [OPTIONS]
polyspace-results-repository -set-sqo-id SQOLevel -product productName -prog
projectName -verif-version versionNumber [OPTIONS]

polyspace-results-repository -delete -product productName -prog projectName -
verif-version versionNumber [OPTIONS]
polyspace-results-repository -rename -product productName -new-prog
newProjectName -new-verif-version newVersionNumber -prog projectName -verif-
version versionNumber [OPTIONS]
```

Description

`polyspace-results-repository -upload resultsFolder -product productName -prog projectName -verif-version versionNumber [OPTIONS]` uploads Polyspace results in `resultsFolder` to the Polyspace Metrics web repository.

You can customize the default upload with additional options.

`polyspace-results-repository -download resultsFolder -product productName -prog projectName -verif-version versionNumber [OPTIONS]` downloads Polyspace results from the Polyspace Metrics web repository to `resultsFolder`.

You can customize the default download with additional options.

`polyspace-results-repository -get-projects-list -product productName` displays the Bug Finder or Code Prover projects currently in the Polyspace Metrics web repository.

`polyspace-results-repository -get-versions-list -product productName -prog projectName` displays the versions of a project currently in the Polyspace Metrics web repository. If the project involves file-by-file verification in Code Prover, add the `-unit-by-unit` option.

`polyspace-results-repository -get-run-numbers-list -product productName -prog projectName -verif-version versionNumber` displays the run numbers of a project version currently in the Polyspace Metrics web repository.

The option is useful only if multiple results with the same project name and version number have been uploaded to Polyspace Metrics.

`polyspace-results-repository -get-files-list -product productName -prog projectName -verif-version versionNumber [OPTIONS]` displays the files involved in the results for a certain project and version.

`polyspace-results-repository -get-sqo-id -product productName -prog projectName -verif-version versionNumber [OPTIONS]` displays the Software Quality Objectives being applied to a certain project and version.

`polyspace-results-repository -set-sqo-id SQOLevel -product productName -prog projectName -verif-version versionNumber [OPTIONS]` applies Software Quality Objectives specified by `SQOLevel` to a certain project and version.

`polyspace-results-repository -delete -product productName -prog projectName -verif-version versionNumber [OPTIONS]` deletes a certain project version from the Polyspace Metrics web repository.

`polyspace-results-repository -rename -product productName -new-prog newProjectName -new-verif-version newVersionNumber -prog projectName -verif-version versionNumber [OPTIONS]` renames a certain project version to another project and version.

Examples

Upload Results to Polyspace Metrics

Suppose you want to upload Code Prover results from the folder `C:\My_Results` to the Polyspace Metrics server `localhost:12427`. You want the project name to appear as `Polyspace_Project` and the version number `1.0`.

Upload the results using this information.

```
polyspace-results-repository -upload "C:\My_Results" \
                             -prog "Polyspace_Project" \
                             -verif-version "1.0" \
                             -server "localhost:12427" \
                             -product "CodeProver"
```

Download Results from Polyspace Metrics

Suppose you want to download Bug Finder results in version `1.0` of the project `Polyspace_Project` from the Polyspace Metrics server `localhost:12427`. You want the results to be downloaded to the folder `C:\My_Results`.

Download the results using this information.

```
polyspace-results-repository -download "C:\My_Results" \
                             -prog "Polyspace_Project" \
                             -verif-version "1.0" \
                             -server "localhost:12427" \
                             -product "BugFinder"
```

Upload Results of Multiple Modules to Polyspace Metrics

If a Polyspace project consists of multiple modules, you can upload the analysis results for all modules to the Polyspace Metrics interface.

For instance, if you run `polyspace-autosar`, a separate module is created for each AUTOSAR Software Component. You can write a shell script (`.sh` file) like this (or a Windows `.bat` file) to collect result files in subfolders of the project folder and upload them to Polyspace Metrics. Code Prover result files use extension `.pscp`.

```
#! /bin/bash
# Upload all results from a polyspace-autosar run to a Metrics server.
MODULES_DIR=`find "$RESULTS_DIR" -name ps_results.pscp -printf '%h\n'`
IFS='
'
for module in $MODULES_DIR; do
    # extract module name from its path foo/bar/behavior_name
    module_name=${module#*AUTOSAR/}
    # transform it to foo.bar.behavior_name
    module_name=${module_name//\//.}
    polyspace-results-repository          \
        -f                                \
        -server localhost                 \
        -upload “$module”                 \
        -prog APPLICATION_NAME            \
        -module $module_name              \
        -verif-version "$RESULTS_VERSION”
done
```

Input Arguments

`resultsFolder` — Folder containing Polyspace results

string

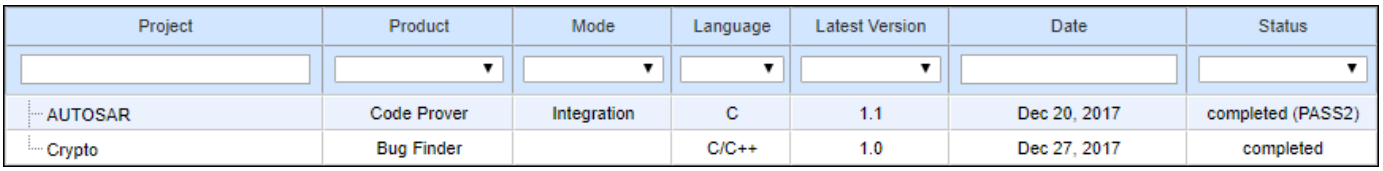
Folder name, specified as a string (in double quotes). The folder must contain a Bug Finder result file (`.psbf`) or a Code Prover file (`.pscp`).

Example: `"C:\Polyspace_Projects\Proj1\Module_1\BF_Result"`, `"C:\AUTOSAR\Demo \polyspace\AUTOSAR\pkg\tst002\swc002\bhv\verification"`

`projectName` — Name of Polyspace project

string

Name of Polyspace project, as it appears on Polyspace metrics.



Project	Product	Mode	Language	Latest Version	Date	Status
AUTOSAR	Code Prover	Integration	C	1.1	Dec 20, 2017	completed (PASS2)
Crypto	Bug Finder		C/C++	1.0	Dec 27, 2017	completed

Example: `"Polyspace_project"`

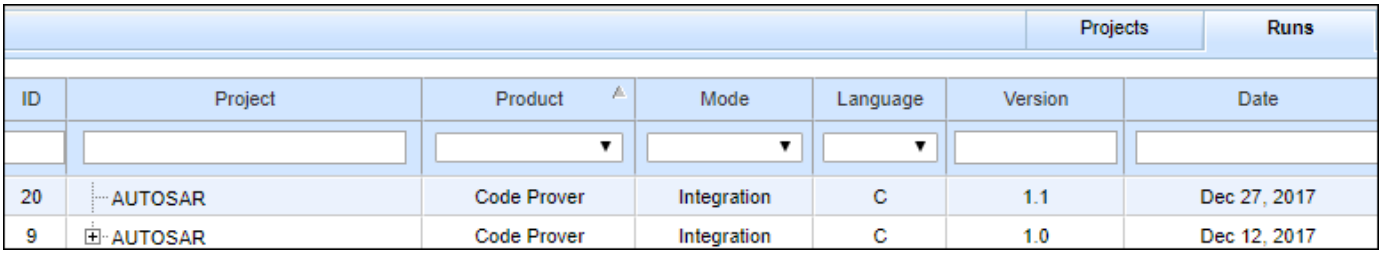
`newProjectName` — Name of Polyspace project
string

New name of Polyspace project, as it appears on Polyspace metrics.

Example: `"Polyspace_project_1"`

`versionNumber` — Version number of Polyspace project
string

Version number of Polyspace project, as it appears on the **Runs** tab of Polyspace metrics.



Projects | Runs

ID	Project	Product	Mode	Language	Version	Date
20	AUTOSAR	Code Prover	Integration	C	1.1	Dec 27, 2017
9	AUTOSAR	Code Prover	Integration	C	1.0	Dec 12, 2017

Example: `"1.0"`

`newVersionNumber` — Version number of Polyspace project
string

New version number of Polyspace project, as it appears on the **Runs** tab of Polyspace metrics.

Example: `"1.1"`

`productName` — Name of product used for analysis
`"CodeProver"` (default) | `"BugFinder"`

Name of product used for producing the results, specified as `"BugFinder"` or `"CodeProver"`.

`SQOLevel` — SQO Level or BF-QO Level to be applied to analysis results
`"SQO-1"` | `"SQO-2"` | `"SQO-3"` | `"SQO-4"` | `"SQO-5"` | `"SQO-6"` | `"BF-QO-1"` | `"BF-QO-2"` | `"BF-QO-3"` | `"BF-QO-4"` | `"BF-QO-5"` | `"BF-QO-6"` | `"Exhaustive"`

Quality levels applied to analysis results. The quality levels consist of a set of criteria based on which the analysis results are assigned a status of **PASS** or **FAIL**. Use the SQO levels for Code Prover results and BF-QO level for Bug Finder results.

See:

- “Software Quality Objectives” (Polyspace Code Prover)
- “Bug Finder Quality Objective Levels”

`[OPTIONS]` — Options to customize upload or download
option name

Option	Description
`-server` *`serverName:portNumber`*	Explicitly specify a server name and port number for upload or download, for instance, `"localhost:12427"`. By default, results are uploaded to or downloaded from the server that you configured in Polyspace preferences. See “Set Up Polyspace Metrics”.
`-f`	Use this option in scripts so that the `polyspace-results-repository` command does not require user interaction. By default, the command asks for confirmation before transferring results from your local folder to Polyspace Metrics or vice versa.
`-password` *`password_value`*	Specify the password for uploading or download a password-protected result in Polyspace Metrics.
`-module` *`module_name`*	Specify that the result belongs to a module in the current Polyspace project. Specify a module name. Use this option to upload results from a project with multiple modules. In Polyspace Metrics, all modules with the same `-prog` value appear under the same project. When you upload the results of multiple modules in the same project, they appear as separate modules in Polyspace Metrics. When you download the result of a specific module, the result appears in a subfolder of the download folder.
`-run-number`	If you uploaded multiple results with the same project name and version number, they appear as separate runs in Polyspace metrics. Use this option to upload or download the results for a specific run.
`-integration` or `-unit-by-unit`	If you run a file-by-file verification, use `-unit-by-unit` to upload or download all results together. Otherwise, use `-integration`. For more information on file-by-file verification, see `Verify files independently (-unit-by-unit)`. By default, the command assumes one result for each upload or download.

Introduced in R2013b

polyspace-comments-import

(DOS/UNIX) Import review information from previous Polyspace analysis

Syntax

```
polyspace-comments-import -diff-rte prevResultsFolder currentResultsFolder
```

Description

`polyspace-comments-import -diff-rte prevResultsFolder currentResultsFolder` imports review information from a results file in `prevResultsFolder` to `currentResultsFolder`. The review information includes the severity, status and additional notes that you assign to a result. Besides importing the review information, the command also shows the number of results where review information could not be imported either because the result changed or the result already had new review information.

Examples

Import Review Information from Previous Polyspace Results

Run Bug Finder on a sample file and add some review information. Then, run Bug Finder a second time and import the information from the previous run.

Copy the file `numerical.c` from *`polyspaceroot`*`\polyspace\examples\cxx\Bug_Finder_Example\sources` to a writable folder. Open a command window and navigate to the folder (using `cd`). Run Bug Finder on the file and save results in the subfolder `Run_1`:

```
polyspace-bug-finder -sources numerical.c -results-dir Run_1/
```

Depending on the product installed, you can also run `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server`.

Open the results file in the `Run_1` subfolder:

```
polyspace Run_1/ps_results.psbf
```

Select a result. On the **Result Details** window, select a **Severity** and **Status** and add some notes. You will import this review information to results from a later analysis.

Run Bug Finder again, but save the results in a different subfolder `Run_2`:

```
polyspace-bug-finder -sources numerical.c -results-dir Run_2/
```

You can open the results file in `Run_2` and see that there is no review information.

Import the review information from the results file in the `Run_1` subfolder to the `Run_2` subfolder:

```
polyspace-comments-import -diff-rte Run_1/ Run_2/
```

Open the results file in the Run_2 subfolder:

```
polyspace Run_2/ps_results.psbf
```

You see the review information imported from the results file in the Run_1 subfolder.

Input Arguments

prevResultsFolder — Folder containing previous Polyspace results with review information

string

Path to a folder containing a Polyspace results file (`.psbf` file for Bug Finder results and `.pscp` file for Code Prover results). The results are presumably from an earlier Polyspace analysis and contain review information that will be imported to a later results file.

Example: `"C:\Polyspace\Project_1_Run_25"`

currentResultsFolder — Folder containing later Polyspace results

string

Path to a folder containing Polyspace results (`.psbf` file for Bug Finder results and `.pscp` file for Code Prover results). The results are presumably from a later Polyspace analysis and have no review information or review information for new results only. You want to import review information from an earlier Polyspace analysis to these results.

Example: `"C:\Polyspace\Project_1_Run_26"`

See Also

`-import-comments`

Topics

“Import Review Information from Previous Polyspace Analysis”

Introduced in R2013b

pslinkfun

Manage model analysis at the command line

Syntax

```
pslinkfun('annotations','type',typeValue,'kind',kindValue,Name,Value)

pslinkfun('openresults',systemName)

pslinkfun('settemplate',psprjFile)
prjTemplate = pslinkfun('gettemplate')

pslinkfun('advancedoptions')
pslinkfun('enablebacktomodel')
pslinkfun('help')
pslinkfun('metrics')
pslinkfun('jobmonitor')
pslinkfun('stop')
```

Description

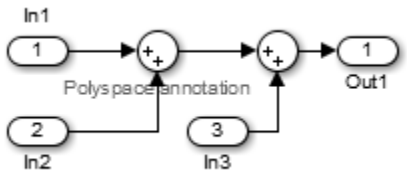
`pslinkfun('annotations','type',typeValue,'kind',kindValue,Name,Value)` adds an annotation of type `typeValue` and kind `kindValue` to the selected block in the model. You can specify a different block using a `Name,Value` pair argument. You can also add notes about a severity classification, an action status, or other comments using `Name,Value` pairs.

In the generated code associated with the annotated block, Polyspace adds code comments before and after the lines of code. Polyspace reads these comments and marks Polyspace results of the specified `kind` with the annotated information.

Syntax limitations:

- You can have only one annotation per block. If a block produces both a rule violation and an error, you can annotate only one type.
- Even though you apply annotations to individual blocks, the scope of the annotation can be larger. The generated code from one block can overlap with another, causing the annotation to also overlap.

  For example, consider this model. The first summation block has a Polyspace annotation, but the second does not.

  However, the associated generated code adds all three inputs in one line of code.

```
/* polyspace:begin<RTE:OVFL:Medium:To Fix>*/
annotate_y.Out1=(annotate_u.In1+annotate_U.In2)+annotate_U.In3;
/* polyspace:end<RTE:OVFL:Medium:To Fix> */
```

Therefore, the annotation justifies both summations.

`pslinkfun('openresults',systemName)` opens the Polyspace results associated with the model or subsystem `systemName` in the Polyspace environment.

`pslinkfun('settemplate',psprjFile)` sets the configuration file for new verifications.

`prjTemplate = pslinkfun('gettemplate')` returns the template configuration file used for new analyses.

`pslinkfun('advancedoptions')` opens the advanced verification options window to configure additional options for the current model.

`pslinkfun('enablebacktomodel')` enables the back-to-model feature of the Simulink plug-in. If your Polyspace results do not properly link to back to the model blocks, run this command.

`pslinkfun('help')` opens the Polyspace documentation in a separate window. Use this option for only pre-R2013b versions of MATLAB.

`pslinkfun('metrics')` opens the Polyspace Metrics interface.

`pslinkfun('jobmonitor')` opens the Polyspace Job Monitor to display remote verifications in the queue.

`pslinkfun('stop')` kills the code analysis that is currently running. Use this option for local analyses only.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

Examples

Annotate a Block and Run a Polyspace Bug Finder Analysis

Use the Polyspace annotation function to annotate a block and see the annotation in the analysis results.

In the example model WhereAreTheErrors, add an annotation to the switch block for MISRA C rule 13.7 violations with a comment, a severity, and a status.

```
model = 'WhereAreTheErrors';
open(model)
pslinkfun('annotations','type','Misra-C', 'kind', '13.7','block',...
    'WhereAreTheErrors/Switch1','status','to fix','comment','must fix')
```

In the open model, you can see a Polyspace annotation added to the Switch block.

Generate code for the model and run an analysis. After the analysis is finished, open the results in the Polyspace environment:

```
slbuild(model)
opts = pslinkoptions(model);
opts.VerificationMode = 'BugFinder';
opts.VerificationSettings = 'PrjConfigAndMisra';
pslinkrun(model,opts)
pslinkfun('openresults',model)
```

The five MISRA C 13.7 rule violations are annotated with the information you added to the switch block. The annotations appear in the **Status** and **Comment** columns.

Add Batch Options to Default Configuration Template

Change advanced Polyspace options and set the new configuration as a template.

Load the model `WhereAreTheErrors` and open the advanced options window.

```
model = 'WhereAreTheErrors';
load_system(model)
pslinkfun('advancedoptions')
```

In the **Run Settings** pane, select the options **Run Bug Finder analysis on a remote cluster** and **Upload results to Polyspace Metrics**.

Set the configuration template for new Polyspace analyses to have these options.

```
pslinkfun('settemplate',fullfile(cd,'pslink_config',...
    'WhereAreTheErrors_config.psprj'))
```

View the current Polyspace template.

```
template = pslinkfun('gettemplate')
```

```
template =
C:\ModelLinkDemo\pslink_config\WhereAreTheErrors_config.psprj
```

View Polyspace Queue and Metrics

Run a remote analysis, view the analysis in the queue, and review the metrics.

Before performing this example, check that your Polyspace configuration is set up for remote analysis and Polyspace Metrics.

Build the model `WhereAreTheErrors`, create a Polyspace options object, set the verification mode, and open the advanced options window.

```
model = 'WhereAreTheErrors';
load_system(model)
slbuild(model)
opts = pslinkoptions(model);
opts.VerificationMode = 'BugFinder';
pslinkfun('advancedoptions')
```

In the **Run Settings** pane, select the options **Run Bug Finder analysis on a remote cluster** and **Upload results to Polyspace Metrics**.

Run Polyspace, then open the Job Monitor to monitor your remote job.

```
pslinkrun(model,opts)
pslinkfun('jobmonitor')
```

After your job is finished, open the metrics server to see your job in the repository.

```
pslinkfun('metrics')
```

Input Arguments

`typeValue` — type of result

`'DEFECT'` | `'MISRA-C'` | `'MISRA-AC-AGC'` | `'MISRA-CPP'` | `'JSF'`

The type of result with which to annotate the block, specified as:

- `'DEFECT'` for defects.
- `'MISRA-C'` for MISRA C coding rule violations (C code only).
- `'MISRA-AC-AGC'` for MISRA C coding rule violations (C code only).
- `'MISRA-CPP'` for MISRA C++ coding rule violations (C++ code only).
- `'JSF'` for JSF C++ coding rule violations (C++ code only).

Example: `'type','MISRA-C'`

`kindValue` — specific check or coding rule

check acronym | rule number

The specific check or coding rule specified by the acronym of the check or the coding rule number. For the specific input for each type of annotation, see the following table.

`type` Value	`kind` Values
`'DEFECT'`	Use the abbreviation associated with the type of defect that you want to annotate. For example, `'int_ovfl'` – Integer overflow. For the list of possible checks, see: “Polyspace Bug Finder Results”.
`'MISRA-C'`	Use the rule number that you want to annotate. For example, `'2.2'`. For the list of supported MISRA C rules and their numbers, see “MISRA C:2004 and MISRA AC AGC Coding Rules”.
`'MISRA-AC-AGC'`	Use the rule number that you want to annotate. For example, `'2.2'`. For the list of supported MISRA C rules and their numbers, see “MISRA C:2004 and MISRA AC AGC Coding Rules”.
`'MISRA-CPP'`	Use the rule number that you want to annotate. For example, `'0-1-1'`. For the list of supported MISRA C++ rules and their numbers, see “MISRA C++:2008 Rules”.
`'JSF'`	Use the rule number that you want to annotate. For example, `'3'`. For the list of supported JSF C++ rules and their numbers, see “JSF C++ Coding Rules”.

Example: `pslinkfun('annotations','type','MISRA-CPP','kind','1-2-3')`

Data Types: `char`

`systemName` — Simulink model
system | subsystem

Simulink model specified by the system or subsystem name.

Example: `pslinkfun('openresults','WhereAreTheErrors')`

`psprjFile` — Polyspace project file
standard Polyspace template (default) | absolute path to `.psprj` file

Polyspace project file specified as the absolute path to the `.psprj` project file. If `psprjFile` is empty, Polyspace uses the standard Polyspace template file. New Polyspace projects start with this project configuration.

Example: `pslinkfun('settemplate', fullfile(polyspaceroot, 'polyspace', 'examples', 'cxx', 'Bug_Finder_Example', 'Bug_Finder_Example.bf.psprj'));`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'block','MyModel\Sum', 'status','to fix'`

`block` — block to be annotated
`gcb` (default) | block name

The block you want to annotate specified by the block name. If you do not use this option, the block returned by the function `gcb` is annotated.

Example: `'block','MyModel\Sum'`

`class` — severity of the check
`'high'` | `'medium'` | `'low'` | `'unset'`

Severity of the check specified as `high`, `medium`, `low`, or `unset`.

Example: `'class','high'`

`status` — action status
`'unreviewed'` | `'to investigate'` | `'to fix'` | `'justified'` | `'no action planned'` | `'not a defect'` | `'other'`

Action status of the check specified as `unreviewed`, `to investigate`, `to fix`, `justified`, `no action planned`, `not a defect`, or `other`.

Example: `'status','no action planned'`

`comment` — additional comments
character vector

Additional comments specified as a character vector. The comments provide more information about why the results are justified.

Example: `'comment','defensive code'`

See Also

`pslinkrun` | `pslinkoptions` | `gcb`

Introduced in R2014a

pslinkoptions

Create options object to customize Polyspace runs from MATLAB command line

Syntax

```
opts = pslinkoptions(codegen)
opts = pslinkoptions(model)
opts = pslinkoptions(sfunc)
```

Description

`opts = pslinkoptions(codegen)` returns an options object with the configuration options for code generated by `codegen`.

`opts = pslinkoptions(model)` returns an options object with the configuration options for the Simulink model.

`opts = pslinkoptions(sfunc)` returns an options object with the configuration options for the S-Function.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

Examples

Use a Simulink model to create and edit an options objects

Load psdemo_model_link_sl and create a Polyspace® options object from the model:

```
load_system('psdemo_model_link_sl');
model_opt = pslinkoptions('psdemo_model_link_sl')
```

```
model_opt =

                    ResultDir: 'results_$ModelName$'
          VerificationSettings: 'PrjConfig'
            OpenProjectManager: 1
          AddSuffixToResultDir: 0
      EnableAdditionalFileList: 0
            AdditionalFileList: {}
              VerificationMode: 'CodeProver'
           EnablePrjConfigFile: 0
                 PrjConfigFile: ''
          AddToSimulinkProject: 0
                InputRangeMode: 'DesignMinMax'
                ParamRangeMode: 'None'
               OutputRangeMode: 'None'
            ModelRefVerifDepth: 'All'
       ModelRefByModelRefVerif: 0
```

```
                AutoStubLUT: 0
      CxxVerificationSettings: 'PrjConfig'
    CheckConfigBeforeAnalysis: 'OnWarn'
```

The model is already configured for Embedded Coder®, so only the Embedded Coder configuration options appear.

Change the results folder name option and set `OpenProjectManager` to true.

```
model_opt.ResultDir = 'results_v1_$ModelName$';
model_opt.OpenProjectManager = true
```

```
model_opt =

                    ResultDir: 'results_v1_$ModelName$'
          VerificationSettings: 'PrjConfig'
            OpenProjectManager: 1
          AddSuffixToResultDir: 0
     EnableAdditionalFileList: 0
           AdditionalFileList: {}
             VerificationMode: 'CodeProver'
          EnablePrjConfigFile: 0
                PrjConfigFile: ''
          AddToSimulinkProject: 0
               InputRangeMode: 'DesignMinMax'
               ParamRangeMode: 'None'
              OutputRangeMode: 'None'
            ModelRefVerifDepth: 'All'
      ModelRefByModelRefVerif: 0
                AutoStubLUT: 0
      CxxVerificationSettings: 'PrjConfig'
    CheckConfigBeforeAnalysis: 'OnWarn'
```

Create and edit an options object for Embedded Coder at the command line

Create a Polyspace® options object called `new_opt` with Embedded Coder parameters:

```
new_opt = pslinkoptions('ec')
```

```
new_opt =

                    ResultDir: 'results_$ModelName$'
          VerificationSettings: 'PrjConfig'
            OpenProjectManager: 0
          AddSuffixToResultDir: 0
     EnableAdditionalFileList: 0
           AdditionalFileList: {}
             VerificationMode: 'CodeProver'
          EnablePrjConfigFile: 0
                PrjConfigFile: ''
          AddToSimulinkProject: 0
               InputRangeMode: 'DesignMinMax'
               ParamRangeMode: 'None'
              OutputRangeMode: 'None'
            ModelRefVerifDepth: 'Current model only'
      ModelRefByModelRefVerif: 0
```

```
                AutoStubLUT: 1
      CxxVerificationSettings: 'PrjConfig'
    CheckConfigBeforeAnalysis: 'OnWarn'
```

To follow the progress in the Polyspace interface, set the `OpenProjectManager` option to true. Change the configuration to check for both checks and MISRA C® 2012 coding rule violations:

```
new_opt.OpenProjectManager = true;
new_opt.VerificationSettings = 'PrjConfigAndMisraC2012'
```

```
new_opt =

                   ResultDir: 'results_$ModelName$'
         VerificationSettings: 'PrjConfigAndMisraC2012'
           OpenProjectManager: 1
         AddSuffixToResultDir: 0
     EnableAdditionalFileList: 0
           AdditionalFileList: {}
             VerificationMode: 'CodeProver'
          EnablePrjConfigFile: 0
                PrjConfigFile: ''
         AddToSimulinkProject: 0
               InputRangeMode: 'DesignMinMax'
               ParamRangeMode: 'None'
              OutputRangeMode: 'None'
        ModelRefVerifDepth: 'Current model only'
      ModelRefByModelRefVerif: 0
                AutoStubLUT: 1
      CxxVerificationSettings: 'PrjConfig'
    CheckConfigBeforeAnalysis: 'OnWarn'
```

Create and edit an options object for TargetLink at the command line

Create a Polyspace® options object called `new_opt` with TargetLink® parameters:

```
new_opt = pslinkoptions('tl')
```

```
new_opt =

                   ResultDir: 'results_$ModelName$'
         VerificationSettings: 'PrjConfig'
           OpenProjectManager: 0
         AddSuffixToResultDir: 0
     EnableAdditionalFileList: 0
           AdditionalFileList: {}
             VerificationMode: 'CodeProver'
          EnablePrjConfigFile: 0
                PrjConfigFile: ''
         AddToSimulinkProject: 0
               InputRangeMode: 'DesignMinMax'
               ParamRangeMode: 'None'
              OutputRangeMode: 'None'
                  AutoStubLUT: 1
```

Set the `OpenProjectManager` option to true to follow the progress in the Polyspace interface. Also change the configuration to check for both run-time errors and MISRA C® coding rule violations:

```
new_opt.OpenProjectManager = true;
new_opt.VerificationSettings = 'PrjConfigAndMisra'

new_opt =

                    ResultDir: 'results_$ModelName$'
          VerificationSettings: 'PrjConfigAndMisra'
            OpenProjectManager: 1
          AddSuffixToResultDir: 0
      EnableAdditionalFileList: 0
            AdditionalFileList: {}
              VerificationMode: 'CodeProver'
           EnablePrjConfigFile: 0
                 PrjConfigFile: ''
          AddToSimulinkProject: 0
                InputRangeMode: 'DesignMinMax'
                ParamRangeMode: 'None'
               OutputRangeMode: 'None'
                   AutoStubLUT: 1
```

Input Arguments

`codegen` — Code generator

`'ec'` | `'tl'`

Code generator, specified as either `'ec'` for Embedded Coder® or `'tl'` for TargetLink®. Each argument creates a Polyspace options object with properties specific to that code generator.

For a description of all configuration options and their values, see pslinkoptions.

Example: `ec_opt = pslinkoptions('ec')`

Example: `tl_opt = pslinkoptions('tl')`

Data Types: `char`

`model` — Simulink model name

model name

Simulink model, specified by the model name. Creates a Polyspace options object with the configuration options of that model. If you have not set any options, the object has the default configuration options. If you have set a code generator, the object has the default options for that code generator.

For a description of all configuration options and their values, see pslinkoptions.

Example: `model_opt = pslinkoptions('my_model')`

Data Types: `char`

`sfunc` — path to S-Function

character vector

Path to S-Function, specified as a character vector. Creates a Polyspace options object with the configuration options for the S-function. If you have not set any options, the object has the default configuration options.

For a description of all configuration options and their values, see pslinkoptions.

Example: `sfunc_opt = pslinkoptions('path/to/sfunction')`

Data Types: `char`

Output Arguments

`opts` — Polyspace configuration options

options object

Polyspace configuration options, returned as an options object. The object is used with `pslinkrun` to run Polyspace from the MATLAB command line.

For the list of object properties, see pslinkoptions.

Example:
```
opts= pslinkoptions('ec')
opts.VerificationSettings = 'Misra'
```

See Also

`pslinkfun` | `pslinkrun`

Topics

pslinkoptions

Introduced in R2012a

pslinkrun

Run Polyspace analysis on model, system, or S-Function

Syntax

```
[polyspaceFolder, resultsFolder] = pslinkrun
[polyspaceFolder, resultsFolder]= pslinkrun(target)
[polyspaceFolder, resultsFolder] = pslinkrun('-slcc',target)
[polyspaceFolder, resultsFolder] = pslinkrun(target, opts)
[polyspaceFolder, resultsFolder] = pslinkrun('-slcc', target, opts)
[polyspaceFolder, resultsFolder] = pslinkrun(target, opts, asModelRef)
[polyspaceFolder, resultsFolder] = pslinkrun('-codegenfolder', codegenFolder,
opts)
```

Description

`[polyspaceFolder, resultsFolder] = pslinkrun` analyzes code generated from the current system using the configuration options associated with the current system. It returns the location of the results folder. The current system is the system returned by the command `bdroot`.

`[polyspaceFolder, resultsFolder]= pslinkrun(target)` analyzes `target` with the configuration options associated with the model containing `target`. Before you run an analysis, you must:

- Generate code for models and subsystems.
- Compile S-Functions.

`[polyspaceFolder, resultsFolder] = pslinkrun('-slcc',target)` runs Polyspace on C/C++ custom code included in C Caller blocks and Stateflow charts in the model.

`[polyspaceFolder, resultsFolder] = pslinkrun(target, opts)` analyzes `target` with the configuration options from the options object `opts`. It returns the location of the results folder.

`[polyspaceFolder, resultsFolder] = pslinkrun('-slcc', target, opts)` runs Polyspace on C/C++ custom code included in C Caller blocks and Stateflow charts in the model. The analysis uses the configuration options from the options object `opts`.

`[polyspaceFolder, resultsFolder] = pslinkrun(target, opts, asModelRef)` uses `asModelRef` to specify which type of generated code to analyze—standalone code or model reference code. This option is useful when you want to analyze only a referenced model instead of an entire model hierarchy.

`[polyspaceFolder, resultsFolder] = pslinkrun('-codegenfolder', codegenFolder, opts)` runs Polyspace on C/C++ code generated from MATLAB code and stored in `codegenFolder`.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

Examples

Analyze Generated Code

Use a Simulink model to generate code, set configuration options, and then run an analysis from the command line.

```
% Generate code from the model WhereAreTheErrors.
model = 'WhereAreTheErrors';
load_system(model);
slbuild(model);

% Create a Polyspace options object from the model.
opts = pslinkoptions(model);

% Set properties that define the Polyspace analysis.
opts.VerificationMode = 'CodeProver';
opts.VerificationSettings = 'PrjConfigAndMisraC2012';

% Run Polyspace using the options object.
[polyspaceFolder, resultsFolder] = pslinkrun(model,opts);
bdclose(model);
```

The results and the corresponding Polyspace project are saved to the `results_WhereAreTheErrors` folder, listed in the `polyspaceFolder` variable. The full path to the results folder is in the `resultsFolder` variable.

Analyze Referenced Model Code

Use a Simulink model to generate model reference code, set configuration options, and then run an analysis from the command line.

```
% Generate code from the model WhereAreTheErrors.
% Treat WhereAreTheErrors as if referenced by another model.
model = 'WhereAreTheErrors';
load_system(model);
slbuild(model,'ModelReferenceCoderTargetOnly');

% Create a Polyspace options object from the model.
opts = pslinkoptions(model);

% Set properties that define the Polyspace analysis.
opts.VerificationMode = 'CodeProver';
opts.VerificationSettings = 'PrjConfigAndMisraC2012';

% Run Polyspace with the options object.
[polyspaceFolder, resultsFolder] = pslinkrun(model,opts,true);
bdclose(model);
```

The results and corresponding Polyspace project are saved to the `results_mr_WhereAreTheErrors` folder, listed in the `polyspaceFolder` variable. The full path to the results folder is in the `resultsFolder` variable.

Reuse Analysis Options for Multiple Models

This example shows how to reuse a subset of options for Polyspace analysis of multiple models. Create a generic options object and specify properties that describe the common options. Associate the generic options object with a model-specific options object. Optionally, set some model-specific options and run the Polyspace analysis.

```
% Generate code from the model WhereAreTheErrors.
model = 'psdemo_model_link_sl';
load_system(model);
slbuild(model);

% Create a generic options object to use for multiple model analyses.
opts = polyspace.ModelLinkOptions();
opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
opts.CodingRulesCodeMetrics.MisraC3Subset = 'all';
opts.MergedReporting.ReportOutputFormat = 'PDF';
opts.MergedReporting.EnableReportGeneration = true;

% Create a model-specific options object.
mlopts = pslinkoptions(model);

% Create a project from the generic options object.
% Associate the project with the model-specific options object.
prjfile = opts.generateProject('model_link_opts');
mlopts.EnablePrjConfigFile = true;
mlopts.PrjConfigFile = prjfile;
mlopts.VerificationMode = 'BugFinder';

% Run Polyspace with the model-specific options object.
[polyspaceFolder, resultsFolder] = pslinkrun(model,mlopts);
bdclose(model);
```

After the analysis completes, results open automatically in the Polyspace interface.

Analyze C/C++ Code Generated from MATLAB Code

This example shows how to analyze C/C++ code generated from MATLAB code.

```
% Generate code
codeName = 'average_filter';
matlabFileName = fullfile(polyspaceroot, 'help',...
    'toolbox','codeprover','examples','matlab_coder','averaging_filter.m');
codegenFolder = fullfile(pwd, 'codegenFolder');
codegen(matlabFileName, '-config:lib', '-c', '-args', ...
   {zeros(1,100,'double')}, '-d', codegenFolder);

% Configure Polyspace analysis
opts = pslinkoptions('ec');
```

```
opts.ResultDir = ['results_',codeName];
opts.OpenProjectManager = 1;

% Run Polyspace
[polyspaceFolder, resultsFolder] = pslinkrun('-codegenfolder', codegenFolder, opts);
```

After the analysis completes, results open automatically in the Polyspace interface.

Input Arguments

`target` — Target of the analysis

`bdroot` (default) | model or system name | path to S-Function block

Target of the analysis specified as a character vector, with the model, system, or S-function in single quotes. The default value is the system returned by `bdroot`.

If you analyze custom code in C Caller blocks and Stateflow charts using `pslinkrun('-slcc',...)`, the argument `target` cannot be an S-Function block.

Example: `[polyspaceFolder, resultsFolder] = pslinkrun('demo')` where `demo` is the name of a model.

Example: `[polyspaceFolder, resultsFolder] = pslinkrun('path/to/sfunction')`

Data Types: `char`

`opts` — Configuration options

options associated with `target` (default) | options object

Configuration options for the analysis, specified as a Polyspace options object. The function `pslinkoptions` creates an options object. You can customize the options object by changing the `pslinkoption` properties.

Example: `pslinkrun('demo', opts_demo)` where `demo` is the name of a model and `opts_demo` is an options object.

`asModelRef` — Indicator for model reference analysis

`false` (default) | `true`

Indicator for model reference analysis, specified as true or false.

- If `asModelRef` is false (default), Polyspace analyzes code that is generated as standalone code. This option is equivalent to choosing **Verify Code Generated For > Model** in the Simulink Polyspace options.
- If `asModelRef` is true, Polyspace analyzes code that is generated as model referenced code. This option is equivalent to choosing **Verify Code Generated For > Referenced Model** in the Simulink Polyspace options. Specifying model reference code indicates that Polyspace must look for the generated code in a different location from the location for standalone code.

Data Types: `logical`

`codegenFolder` — Folder containing generated C/C++ code

character vector

Folder containing C/C++ code generated from MATLAB code, specified as a character vector. You specify this folder with the `codegen` command using the flag `-d`.

Output Arguments

`polyspaceFolder` — Folder containing Polyspace project and results

character vector

Name of the folder containing Polyspace project and results, specified as a character vector. The default value of this variable is `results_$ModelName$`.

To change this value, see “Output folder” on page 14-16.

`resultsFolder` — Full path to subfolder containing Polyspace results

character vector

Full path to subfolder containing Polyspace results, specified as a character vector.

The folder `results_$ModelName$` contains your Polyspace project and a subfolder `$ModelName$` with the analysis results. This variable gives you the full path to the subfolder. You can use this path with the `polyspace.BugFinderResults` class.

To change the parent folder `results_$ModelName$`, see “Output folder” on page 14-16.

See Also

`pslinkfun` | `pslinkoptions` | pslinkoptions

Topics

“Run Polyspace Analysis on Code Generated from Simulink Model”
“Run Polyspace Analysis on S-Function Code”
“Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts”
“Recommended Model Configuration Parameters for Polyspace Analysis”

Introduced in R2012a

polyspaceBugFinder

Run Polyspace Bug Finder analysis from MATLAB

Note For easier scripting, run Polyspace® analysis using a `polyspace.Project` object.

Syntax

```
polyspaceBugFinder
polyspaceBugFinder(projectFile)

polyspaceBugFinder(optsObject)
polyspaceBugFinder(projectFile, '-nodesktop')

polyspaceBugFinder(resultsFile)
polyspaceBugFinder('-results-dir',resultsFolder)

polyspaceBugFinder('-help')

polyspaceBugFinder('-sources',sourceFiles)
polyspaceBugFinder('-sources',sourceFiles,Name,Value)
```

Description

`polyspaceBugFinder` opens Polyspace Bug Finder.

`polyspaceBugFinder(projectFile)` opens a Polyspace project file in Polyspace Bug Finder.

`polyspaceBugFinder(optsObject)` runs an analysis on the Polyspace options object in MATLAB.

`polyspaceBugFinder(projectFile, '-nodesktop')` runs an analysis on the Polyspace project file in MATLAB.

`polyspaceBugFinder(resultsFile)` opens a Polyspace results file in Polyspace Bug Finder.

`polyspaceBugFinder('-results-dir',resultsFolder)` opens a Polyspace results file from `resultsFolder` in Polyspace Bug Finder.

`polyspaceBugFinder('-help')` displays options that can be supplied to the `polyspaceBugFinder` command to run a Polyspace Bug Finder analysis.

`polyspaceBugFinder('-sources',sourceFiles)` runs a Polyspace Bug Finder analysis on the source files specified in `sourceFiles`.

`polyspaceBugFinder('-sources',sourceFiles,Name,Value)` runs a Polyspace Bug Finder analysis on the source files with additional options specified by one or more `Name,Value` pair arguments.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

Examples

Open Polyspace Projects from MATLAB

This example shows how to open a Polyspace project file with extension `.psprj` from MATLAB. In this example, you open the project file `Bug_Finder_Example.psprj` from the folder *`polyspaceroot`*`\polyspace\examples\cxx\Bug_Finder_Example`.

Open the project `Bug_Finder_Example.psprj` in the Polyspace interface.

```
prjFile = fullfile(polyspaceroot, 'polyspace', 'examples', 'cxx', ...
         'Bug_Finder_Example', 'Bug_Finder_Example.psprj');
polyspaceBugFinder(prjFile);
```

Open Polyspace Results from MATLAB

This example shows how to open a Polyspace results file from MATLAB. In this example, you open the results file from the folder *`polyspaceroot`*`\polyspace\examples\cxx\Bug_Finder_Example\Module_1\BF_Result`.

Open the results of `resFolder`.

```
resFolder = fullfile(polyspaceroot, 'polyspace', 'examples',  ...
         'cxx', 'Bug_Finder_Example', 'Module_1', 'BF_Result');
polyspaceBugFinder('-results-dir',resFolder)
```

Run Polyspace Analysis with Options Object

This example shows how to run a Polyspace analysis from the MATLAB command-line using objects.

Create an options object and add the source file and include folder to the properties.

```
opts = polyspace.BugFinderOptions;
opts.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
opts.EnvironmentSettings.IncludeFolders = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Bug_Finder_Example', 'sources')};
opts.ResultsDir = fullfile(pwd,'results');
```

Run the analysis and view the results.

```
polyspaceBugFinder(opts);
polyspaceBugFinder('-results-dir',opts.ResultsDir)
```

Run Polyspace Analysis from MATLAB with DOS/UNIX Options

This example shows how to run a Polyspace analysis in MATLAB using DOS/UNIX-style options.

Run the analysis and open the results.

```
sourceFiles = fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c');
includeFolders = fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Bug_Finder_Example', 'sources');
resultsDir = fullfile(pwd,'results');
polyspaceBugFinder('-sources',sourceFiles, ...
             '-I',includeFolders, ...
             '-results-dir',resultsDir);
```

To view the results, enter:

```
polyspaceBugFinder('-results-dir',resultsDir);
```

Run Polyspace Analysis with Coding Rules Checking

This example shows two different ways to customize an analysis in MATLAB. You can customize as many additional options as you want by changing properties in an options object or by using Name-Value pairs. Here you specify checking of MISRA C 2012 coding rules.

Create variables to save the source file path and results folder path. You can use these variables for either analysis method.

```
sourceFileName = fullfile(polyspaceroot, 'polyspace','examples', 'cxx', ...
    'Bug_Finder_Example','sources','dataflow.c');
resFolder1 = fullfile('Polyspace_Results_1');
resFolder2 = fullfile('Polyspace_Results_2');
```

Analyze coding rules with an options object.

```
opts = polyspace.BugFinderOptions();
opts.Sources = {sourceFileName};
opts.ResultsDir = resFolder1;
opts.CodingRulesCodeMetrics.MisraC3Subset = 'all';
opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
polyspaceBugFinder(opts);
polyspaceBugFinder('-results-dir',resFolder1);
```

Analyze coding rules with DOS/UNIX options.

```
polyspaceBugFinder('-sources',sourceFileName,'-results-dir',resFolder2,...
    '-misra3','all');
polyspaceBugFinder('-results-dir',resFolder2);
```

Input Arguments

`optsObject` — Polyspace options object name

object handle

Polyspace options object name, specified as the object handle.

To create an options object, use one of the Polyspace options classes.

Example: `opts`

`projectFile` — Name of `.psprj` file

character vector

Name of project file with extension `.psprj`, specified as a character vector.

If the file is not in the current folder, `projectFile` must include a full or relative path.

Example: `'C:\Polyspace_Projects\myProject.psprj'`

Data Types: `char`

`resultsFile` — Name of `.psbf` file
character vector

Name of results file with extension `.psbf`, specified as a character vector.

If the file is not in the current folder, `resultsFile` must include a full or relative path.

Example: `'myResults.psbf'`

Data Types: `char`

`resultsFolder` — Name of result folder
character vector

Name of result folder, specified as a character vector. The folder must contain the results file with extension `.psbf`. If the results file resides in a subfolder of the specified folder, this command does not open the results file.

If the folder is not in the current folder, `resultsFolder` must include a full or relative path.

Example: `'C:\Polyspace\Results\'`

Data Types: `char`

`sourceFiles` — Comma-separated names of C or C++ files
character vector

Comma-separated C or C++ source file names, specified as a single character vector.

If the files are not in the current folder, `sourceFiles` must include a full or relative path.

Example: `'myFile.c'`, `'C:\mySources\myFile1.c,C:\mySources\myFile2.c'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'-target','i386','-compiler','gnu4.6'` specifies that the source code is intended for a i386 target and contains non-ANSI C syntax for GCC 4.6.

For option names and values, see the **Command-Line Information** section in “Analysis Options”.

See Also

`polyspace.BugFinderOptions` | `polyspace.ModelLinkBugFinderOptions`

Topics
“Run Polyspace Analysis by Using MATLAB Scripts”

Introduced in R2013b

polyspaceConfigure

Create Polyspace project from your build system at the MATLAB command line

Syntax

```
polyspaceConfigure buildCommand

polyspaceConfigure -option value buildCommand
```

Description

`polyspaceConfigure buildCommand` traces your build system and creates a Polyspace project with information gathered from your build system. You can run an analysis on a Polyspace project only in the user interface of the Polyspace desktop products.

`polyspaceConfigure -option value buildCommand` traces your build system and uses `-option value` to modify the default operation of `polyspaceConfigure`. Specify the modifiers before `buildCommand`, otherwise they are considered as options in the build command itself.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

Examples

Create Polyspace Project from Makefile

This example shows how to create a Polyspace project if you use the command `make` *`targetName buildOptions`* to build your source code. The example creates a Polyspace project that can be opened only in the user interface of the Polyspace desktop products.

Create a Polyspace project specifying a unique project name. Use the `-B` or `-W` *`makefileName`* option with `make` so that the all prerequisite targets in the makefile are remade.

```
polyspaceConfigure  -prog myProject ...
          make -B targetName buildOptions
```

Open the Polyspace project in the **Project Browser**.

```
polyspaceBugFinder('myProject.psprj')
```

Create Projects That Have Different Source Files from Same Build Trace

This example shows how to create different Polyspace projects from the same trace of your build system. You can specify which source files to include for each project. The example creates a Polyspace project that can be opened only in the user interface of the Polyspace desktop products.

Trace your build system without creating a Polyspace project by specifying the option `-no-project`. To ensure that all the prerequisite targets in your makefile are remade, use the appropriate `make` build command option, for instance `-B`.

```
polyspaceConfigure -no-project make -B;
```

`polyspace-configure` stores the cache information and the build trace in default locations inside the current folder. To store the cache information and build trace in a different location, specify the options `-cache-path` and `-build-trace`.

Generate Polyspace projects by using the build trace information from the previous step. Specify a project name and use the `-include-sources` or `-exclude-sources` option to select which files to include for each project.

```
polyspaceConfigure -no-build -prog myProject ...
-include-sources "glob_pattern";
```

`glob_pattern` is a glob pattern that corresponds to folders or files you filter in or out of your project. To ensure the shell does not expand the glob patterns you pass to `polysapce-configure`, enclose them in double quotes.For more information on the supported syntax for glob patterns, see “polyspace-configure Source Files Selection Syntax”.

If you specified the options `-build-trace` and `-cache-path` in the previous step, specify them again.

Delete the trace file and cache folder.

```
rmdir('polyspace_configure_cache', 's');
delete polyspace_configure_built_trace;
```

If you used the options `-build-trace` and `-cache-path`, use the paths and file names from those options.

Run Command-Line Polyspace Analysis from Makefile

This example shows how to run Polyspace analysis if you use a build command such as `make` *`targetName buildOptions`* to build your source code. In this example, you use `polyspaceConfigure` to trace your build system but do not create a Polyspace project. Instead you create an options file that you can use to run Polyspace analysis from the command-line.

Create a Polyspace options file specifying the `-output-options-file` command. Use the `-B` or `-W` *`makefileName`* option with `make` so that all prerequisite targets in the makefile are remade.

```
polyspaceConfigure -output-options-file ...
         myOptions make -B targetName buildOptions
```

Use the options file that you created to run a Polyspace analysis at the command line:

```
polyspaceBugFinder -options-file myOptions
```

Input Arguments

`buildCommand` — Command for building source code

build command

Build command specified exactly as you use to build your source code.

Example: `make -B`, `make -W` *`makefileName`*

`-option value` — Options for changing default operation of `polyspaceConfigure`
single option starting with `-`, followed by argument | multiple space-separated option-argument pairs

Basic Options

Option	Argument	Description
`-prog`	Project name	Project name that appears in the Polyspace user interface. The default is `polyspace`. If you do not use the option `-output-project`, the `-prog` argument also sets the project name. **Example:** `-prog myProject` creates a project that has the name `myProject` in the user interface. If you do not use the option `-output-project`, the project name is also `myProject.psrprj`.
`-author`	Author name	Name of project author. **Example:** `-author jsmith`
`-output-project`	Path	Project file name and location for saving project. The default is the file `polyspace.psprj` in the current folder. **Example:** `-output-project ../myProjects/project1` creates a project `project1.psprj` in the folder with the relative path `../myProjects/`.
`-output-options-file`	File name	Option to create a Polyspace analysis options file. Use this file for command-line analysis using `polyspace-bug-finder`.

`-allow-build-error`	None	Option to create a Polyspace project even if an error occurs in the build process.
		
		If an error occurs, the build trace log shows the following message:
		
		`polyspace-configure (polyspaceConfigure)`
		`ERROR: build command`
		`*command_name* fail [status=*status_value*]`
		
		`command_name` is the build command name that you use and *`status_value`* is the non-zero exit status or error level that indicates which error occurred in your build process.

`-allow-overwrite`	None	Option to overwrite a project with the same name, if it exists. By default, `polyspace-configure (polyspaceConfigure)` throws an error if a project with the same name already exists in the output folder. Use this option to overwrite the project.

Option	Argument	Description
`-silent` (default) `-verbose`	None	Option to suppress or display additional messages from running `polyspace-configure (polyspaceConfigure)`.
`-help`	None	Option to display the full list of `polyspace-configure (polyspaceConfigure)` commands
`-debug`	None	Option to store debug information for use by MathWorks technical support. This option has been superseded by the option `-easy-debug`.
`-easy-debug`	Path	Option to store debug information for use by MathWorks technical support. After a `polyspace-configure (polyspaceConfigure)` run, the path provided contains a zipped file ending with `pscfg-output.zip`. If the run fails to create a complete Polyspace project or options file, send this zipped file to MathWorks Technical Support for further debugging. The zipped file does not contain source files traced in the build. See also “Errors in Project Creation from Build Systems”.

Options to Create Multiple Modules

Option	Argument	Description
`-module`	None	Option to create a separate options file for each binary created in build system. You can only create separate options files for different binaries. You cannot create multiple modules in a Polyspace project (for running in the Polyspace user interface). Use this option only for build systems that use GNU and Visual C++ compilers. See also “Modularize Polyspace Analysis by Using Build Command”.
`-output-options-path`	Path name	Location where generated options files are saved. Use this option together with the option `-module`. The options files are named after the binaries created in the build system.

Advanced Options

Option	Argument	Description
`-compiler-config`	Path and file name	Location and name of compiler configuration file. The file must be in a specific format. For guidance, see the existing configuration files in *polyspaceroot*`\polyspace\configure\compiler_configuration\`. For information on the contents of the file, see “Compiler Not Supported for Project Creation from Build Systems”. **Example:** `-compiler-configuration myCompiler.xml`
`-no-project`	None	Option to trace your build system without creating a Polyspace project and save the build trace information. Use this option to save your build trace information for a later run of `polyspace-configure (polyspaceConfigure)` with the `-no-build` option.
`-no-build`	None	Option to create a Polyspace project using previously saved build trace information. To use this option, you must have the build trace information saved from an earlier run of `polyspace-configure (polyspaceConfigure)` with the `-no-project` option. If you use this option, you do not need to specify the `buildCommand` argument.

Option	Argument	Description

`-no-sources`	None	Option to create a Polyspace options file that does not contain the source file specifications.
		
		Use this option when you intend to specify the source files by other means. For instance, you can use this option when:
		
		• Running Polyspace on AUTOSAR-specific code.
		
		You want to create an options file that traces your build command for the compiler options:
		
		`-output-options-file options.txt -no-sources`
		
		You later append this options file when extracting source file names from ARXML specifications and running the subsequent Code Prover analysis with `polyspace-autosar`
		
		`-extra-options-file options.txt`
		
		See also “Run Polyspace on AUTOSAR Code Using Build Command” (Polyspace Code Prover).
		• Running Polyspace in Eclipse.
		
		Your source files are already specified in your Eclipse project. When running a Polyspace analysis, you want to specify an options file that has the compilation options only.

Option	Argument	Description
`-extra-project-options`	Options to use for subsequent Polyspace analysis. For instance, `"-stubbed-pointers-are-unsafe"`.	Options that are used for subsequent Polyspace analysis. Once a Polyspace project is created, you can change some of the default options in the project. Alternatively, you can pass these options when tracing your build command. The flag `-extra-project-options` allows you to pass additional options. Specify multiple options in a space separated list, for instance `"-allow-negative-operand-in-shift -stubbed-pointers-are-unsafe"`. Suppose you have to set the option `-stubbed-pointers-are-unsafe` for every Polyspace project created. Instead of opening each project and setting the option, you can use this flag when creating the Polyspace project: `-extra-project-options "-stubbed-pointers-are-unsafe"` For the list of options available, see: • “Analysis Options” • • If you are creating an options file instead of a Polyspace project from your build command, do not use this flag.
`-tmp-path`	Path	Location of folder where temporary files are stored.
`-build-trace`	Path and file name	Location and name of file where build information is stored. The default is `./polyspace_configure_build_trace.log`. **Example:** `-build-trace ../build_info/trace.log`

`-include-sources`
 	
`-exclude-sources`
	Glob pattern	Option to specify which source files `polyspace-configure (polyspaceConfigure)` includes in, or excludes from, the generated project. You can combine both options together.
		
		A source file is included if the file path matches the glob pattern that you pass to `-include-sources`.
		
		A source file is excluded if the file path matches the glob pattern that you pass to `-exclude-sources`.

Option	Argument	Description

`-print-included-sources`
 	
`-print-excluded-sources`
	None	Option to print the list of source files that `polyspace-configure (polyspaceConfigure)` includes in, or excludes from, the generated project. You can combine both options together. The output displays the full path of each file on a separate line.
		
		Use this option to troubleshoot the glob patterns that you pass to `-include-sources` or `-exclude-sources`. You can see which files match the pattern that you pass to `-include-sources` or `-exclude-sources`.

Cache Control Options

These options are primarily useful for debugging. Use the options if `polyspace-configure (polyspaceConfigure)` fails and MathWorks Technical Support asks you to use the option and provide the cached files. Starting R2020a, the option `-easy-debug` provides an easier way to provide debug information. See “Contact Technical Support About Issues with Running Polyspace”.

Option	Argument	Description

`-no-cache`
 	
`-cache-sources` (default)
 	
`-cache-all-text`
 	
`-cache-all-files`
	None	Option to perform one of the following:
		
		• `-no-cache`: Not create a cache
		• `-cache-sources`: Cache text files temporarily created during build for later use by `polyspace-configure (polyspaceConfigure)`.
		• `-cache-all-text`: Cache all text files including sources and headers.
		• `-cache-all-files`: Cache all files including binaries.
		
		Typically, you cache temporary files created by your build command to debug issues in tracing the command.

`-cache-path`	Path	Location of folder where cache information is stored. **Example:** `-cache-path ../cache`
`-keep-cache` `-no-keep-cache` (default)	None	Option to preserve or clean up cache information after `polyspace-configure (polyspaceConfigure)` completes execution. If `polyspace-configure (polyspaceConfigure)` fails, you can provide this cache information to technical support for debugging purposes.

See Also

Topics
“Modularize Polyspace Analysis by Using Build Command”
“Requirements for Project Creation from Build Systems”
“Compiler Not Supported for Project Creation from Build Systems”

Introduced in R2013b

polyspaceJobsManager

Manage Polyspace jobs on a MATLAB Parallel Server cluster

Syntax

```
polyspaceJobsManager('listjobs')
polyspaceJobsManager('cancel','-job',jobNumber)
polyspaceJobsManager('remove','-job',jobNumber)
polyspaceJobsManager('getlog','-job',jobNumber)
polyspaceJobsManager('wait','-job',jobNumber)
polyspaceJobsManager('promote','-job',jobNumber)
polyspaceJobsManager('demote','-job',jobNumber)

polyspaceJobsManager('download','-job',jobNumber)
polyspaceJobsManager('download','-job',jobNumber,'-results-folder',
resultsFolder)

polyspaceJobsManager(___,'-scheduler',scheduler)
```

Description

`polyspaceJobsManager('listjobs')` lists all Polyspace jobs in your cluster.

`polyspaceJobsManager('cancel','-job',jobNumber)` cancels the specified job. The job appears in your queue as cancelled.

`polyspaceJobsManager('remove','-job',jobNumber)` removes the specified job from your cluster.

`polyspaceJobsManager('getlog','-job',jobNumber)` displays the log for the specified job.

`polyspaceJobsManager('wait','-job',jobNumber)` pauses until the specified job is done.

`polyspaceJobsManager('promote','-job',jobNumber)` moves the specified job up in the MATLAB job scheduler queue.

`polyspaceJobsManager('demote','-job',jobNumber)` moves the specified job down in the MATLAB job scheduler queue.

`polyspaceJobsManager('download','-job',jobNumber)` downloads the results from the specified job. The results are downloaded to the folder you specified when starting analysis, using the `-results-dir` on page 2-35 option.

`polyspaceJobsManager('download','-job',jobNumber,'-results-folder',resultsFolder)` downloads the results from the specified job to `resultsFolder`.

`polyspaceJobsManager(___,'-scheduler',scheduler)` performs the specified action on the job scheduler specified. If you do not specify a server with any of the previous syntaxes, Polyspace uses the server stored in your Polyspace preferences.

Examples

Manipulate Two Jobs in the Cluster

In this example, use a MATLAB Job Scheduler scheduler to run Polyspace remotely and monitor your jobs through the queue.

Before performing this example, set up an MATLAB Job Scheduler and Polyspace Metrics. This example uses the *myMJS@myCompany.com* scheduler. When you perform this example, replace this scheduler with your own cluster name.

Set up your source files.

```
tempDir = fullfile(tempdir, 'psdemo', 'src');
mkdir(tempDir);
demo = fullfile(polyspaceroot,'polyspace','examples','cxx',...
'Bug_Finder_Example','sources');
copyfile(demo,tempDir);
```

Submit two jobs to your scheduler.

If your jobs have not started running, promote the second job to run before the first job.

```
polyspaceJobsManager('promote','-job','20','-scheduler',...
    'myMJS@myCompany.com')
```

Job 20 starts running before job 19.

Cancel job 19.

```
polyspaceJobsManager('cancel','-job','19','-scheduler',...
   'myMJS@myCompany.com')
polyspaceJobsManager('listjobs','-scheduler','myMJS@myCompany.com')
```

Remove job 19.

```
polyspaceJobsManager('remove','-job','19','-scheduler',...
   'myMJS@myCompany.com')
polyspaceJobsManager('listjobs','-scheduler','myMJS@myCompany.com')
```

Get the log for job 20.

```
polyspaceJobsManager('getlog','-job','20','-scheduler',...
   'myMJS@myCompany.com')
```

Download the information from job 20.

```
resFolder3 = fullfile(tempDir, 'res3');
polyspaceJobsManager('download','-job','20','-results-folder', ...
   resFolder3,'-scheduler','myCluster')
```

Input Arguments

`jobNumber` — Queued job number
character vector of job number

Number of the queued job that you want to manage, specified as a character vector in single quotes.

Example: `'-job','10'`

`resultsFolder` — Path to results folder
character vector

Path to results folder specified as a character vector in single quotes. This folder stores the downloaded results files.

Example: `'-results-folder','C:\psdemo\myresults'`

`scheduler` — job scheduler
head node of your cluster | job scheduler name | cluster profile

Job scheduler for remote verifications specified as one of the following:

- Name of the computer that hosts the head node of your MATLAB Parallel Server cluster (*`NodeHost`*).
- Name of the MATLAB Job Scheduler on the head node host (*`MJSName`*`@`*`NodeHost`*).
- Name of a MATLAB cluster profile (*`ClusterProfile`*).

Example: `'-scheduler','myscheduler@mycompany.com'`

See Also

`polyspaceBugFinder`

Topics
“Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox)
“Send Polyspace Analysis from Desktop to Remote Servers Using Scripts” (Polyspace Code Prover)

Introduced in R2013b

polyspaceroot

Get Polyspace installation folder

Syntax

```
polyspaceroot
```

Description

`polyspaceroot` returns the Polyspace installation folder.

Starting in R2019a, to run MATLAB scripts for Polyspace analysis, you install MATLAB and Polyspace in separate folders and link between them. After installation and linking, to access files in the Polyspace installation folder from MATLAB, use this function. See also “Integrate Polyspace with MATLAB and Simulink”.

Examples

Get Polyspace Installation Folder

To determine the Polyspace installation folder, use the `polyspaceroot` function.

```
polyspaceroot
```

```
C:\Program Files\Polyspace\R2019a
```

With the products, Polyspace Bug Finder Server or Polyspace Code Prover Server, the default installation folder in Windows is:

```
C:\Program Files\Polyspace Server\R2019a
```

Run Polyspace on Sample Files in Polyspace Installation Folder

To access sample files in the Polyspace installation folder, use the `polyspaceroot` function to get the root of the installation folder. Append subfolders to the root folder path with the `fullfile` function.

Run Bug Finder on the file `numerical.c` in the subfolder `polyspace\examples\cxx \Bug_Finder_Example\sources` of the Polyspace installation folder.

```
proj = polyspace.Project

% Specify sources and includes
sourceFile = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c');
includeFolder = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');

% Configure analysis
proj.Configuration.Sources = {sourceFile};
```

```
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.EnvironmentSettings.IncludeFolders = {includeFolder};
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = proj.run('bugFinder');
```

See Also

`polyspace.Project`

Topics

“Run Polyspace Analysis by Using MATLAB Scripts”

Introduced in R2019a

polyspace_report

Generate reports from Polyspace analysis results

Syntax

```
polyspace_report('-template', template, '-results-dir', resultsFolder,
options)
polyspace_report('-generate-results-list-file', '-results-dir',
resultsFolder, options)
polyspace_report('-generate-variable-access-file', '-results-dir',
resultsFolder, options)
```

Description

`polyspace_report('-template', template, '-results-dir', resultsFolder, options)` generates a report using a predefined template specified by `template`. By default, the report is named after the results file in the folder `resultsFolder` and saved in the `Polyspace-Doc` subfolder. You can change the default behavior using additional options.

`polyspace_report('-generate-results-list-file', '-results-dir', resultsFolder, options)` exports the list of Polyspace results to a tab-delimited text file.

`polyspace_report('-generate-variable-access-file', '-results-dir', resultsFolder, options)` exports the list of global variables to a tab-delimited text file.

Note

- Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.
- You need MATLAB Report Generator™ to use this function.

Examples

Generate PDF Report from Results

Generate a PDF report from sample Polyspace Code Prover results.

```
template = fullfile(polyspaceroot,'toolbox','polyspace','psrptgen','templates',...
    'Developer.rpt');
resPath = fullfile(polyspaceroot,'polyspace','examples','cxx','Code_Prover_Example',...
    'Module_1','CP_Result');
polyspace_report('-template', template, '-results-dir', resPath, '-format', 'PDF');
```

Input Arguments

`template` — Path to report template file

character vector

Path to report template file, specified as a character vector. To generate multiple reports, specify a comma-separated list of report template paths in the character vector (do not put a space after the commas). The templates are available in *`polyspaceroot`*`\toolbox\polyspace\psrptgen\templates\` as `.rpt` files. Here, *`polyspaceroot`* is the Polyspace installation folder. For more information on the available templates, see `Bug Finder and Code Prover report (-report-template)`.

Example: `fullfile(polyspaceroot,'toolbox','polyspace','psrptgen','templates','Developer.rpt');`

`resultsFolder` — Folder containing analysis results
character vector

Folder containing analysis results, specified as a character vector. The folder must contain a `.psbf` file containing Polyspace Bug Finder results or a `.pscp` file containing Polyspace Code Prover results.

To generate reports for multiple analyses, specify a comma-separated list of folder paths (do not put a space after the commas).

Example: `'C:\Polyspace_Workspace\My_project\Module_1\results'`

`options` — Options for generating report
character vector

Options to control report generation, for instance, output format and output name.

Specify each option as a character vector, followed by the option value as a separate character vector. For instance, you can specify the PDF format by using the syntax `polyspace_report(..., '-format','PDF')`.

Option	Value	Description
`'-format'`	`'PDF'`, `'HTML'` or `'WORD'`	File format of the report that you generate. By default, the command generates a Word document. To generate reports in multiple formats, specify a comma-separated list of formats. (Do not put a space after the commas). For instance, `polyspace_report(..., '-format', 'PDF,HTML')`. This option is not compatible with `-generate-variable-access-file` and `-generate-results-list-file`.
`'-set-language-english'`		Generate the report in English. Use this option if your display option is set to another language.

Option	Value	Description
`'-output-name'`	Report name, for instance, `PolyspaceReport`.	Name of the generated report or folder name if you generate multiple reports. The full path to the report is created by appending the name to the current working folder. To store the reports on a different path, specify the full path as value for this option.

See Also

Introduced in R2013b

polyspace.Project

Run Polyspace analysis on C and C++ code and read results

Description

Run a Polyspace analysis on C and C++ source files by using this MATLAB object. To specify source files and customize analysis options, use the `Configuration` property. To run the analysis, use the `run` method. To read results after analysis, use the `Results` property.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

Creation

`proj = polyspace.Project` creates an object that you can use to configure and run a Polyspace analysis, and then read the analysis results.

Properties

`Configuration` — Analysis options
`polyspace.Options` object

Options for running Polyspace analysis, implemented as a `polyspace.Options` object. The object has properties corresponding to the analysis options. For more information on those properties, see polyspace.Project.Configuration properties.

You can retain the default options or change them in one of these ways:

- Set the source code language to `'C'`, `'CPP'`, or `'C-CPP'` (default). Some analysis options might not be available depending on the language setting of the object.

  ```
  proj=polyspace.Project;
  proj.Configuration=polyspace.Options('C');
  ```

- Modify the properties directly.

  ```
  proj = polyspace.Project;
  proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
  ```

- Obtain the options from another `polyspace.Project` object.

  ```
  proj1 = polyspace.Project;
  proj1.Configuration.TargetCompiler.Compiler = 'gnu4.9';

  proj2 = proj1;
  ```

  To use common analysis options across multiple projects, follow this approach. For instance, you want to reuse all options and change only the source files.

- Obtain the options from a project created in the user interface of the Polyspace desktop products (`.psprj` file).

```
proj = polyspace.Project;
projectLocation = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'Bug_Finder_Example.psprj')
proj.Configuration = polyspace.loadProject(projectLocation);
```

To determine the optimal set of options, set your options in the user interface and then import them to a `polyspace.Project` object. In the user interface, you can access help from features such as the Compilation Assistant and get tooltip help on options.

- Obtain the options from a Simulink model (applies only to Polyspace desktop products). Before obtaining the options, generate code from the model.

```
modelName = 'rtwdemo_roll';
load_system(modelName);

% Set parameters for Embedded Coder target
set_param(modelName, 'SystemTargetFile', 'ert.tlc');
set_param(modelName,'Solver','FixedStepDiscrete');
set_param(modelName,'SupportContinuousTime','on');
set_param(modelName,'LaunchReport','off');
set_param(modelName,'InitFltsAndDblsToZero','on');

if exist(fullfile(pwd,'rtwdemo_roll_ert_rtw'), 'dir') == 0
    rtwbuild(modelName);
end

% Obtain configuration from model
proj = polyspace.Project;
proj.Configuration = polyspace.ModelLinkOptions(modelName);
```

Use the options to analyze the code generated from the model.

`Results` — Analysis results
`polyspace.BugFinderResults` or `polyspace.CodeProverResults` object

Results of Polyspace analysis. When you create a `polyspace.Project` object, this property is initially empty. The property is populated only after you execute the `run` method of the object. Depending on the argument to the `run` method, `'bugFinder'` or `'codeProver'`, the property is implemented as a `polyspace.BugFinderResults` or `polyspace.CodeProverResults` object.

To read the results, use these methods of the `polyspace.BugFinderResults` or `polyspace.CodeProverResults` object:

- `getSummary`: Obtain a summarized format of the results into a MATLAB table.

```
proj = polyspace.Project;
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.ResultsDir = fullfile(pwd,'results');

run(proj, 'bugFinder');

resTable = proj.Results.getSummary('defects');
```

For more information, see `getSummary` or `getSummary`.

- `getResults`: Obtain the full results or a more readable format into a MATLAB table.

```
proj = polyspace.Project;
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.ResultsDir = fullfile(pwd,'results');

run(proj, 'bugFinder');

resTable = proj.Results.getResults('readable');
```

For more information, see `getResults` or `getResults`.

Object Functions

run Run a Polyspace analysis

Examples

Check for Bugs

Run a Polyspace Bug Finder analysis on the example file `numerical.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = run(proj, 'bugFinder');

% Read results
bfSummary = proj.Results.getSummary('defects');
```

Prove Absence of Run-Time Errors

Run a Polyspace Code Prover analysis on the example file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a `main` function must be generated, if the function does not exist in the source code.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');
proj.Configuration.CodeProverVerification.MainGenerator = true;
```

```
% Run analysis
cpStatus = run(proj, 'codeProver');

% Read results
cpSummary = proj.Results.getSummary('runtime');
```

Check for Bugs and MISRA C:2012 Violations

Run a Polyspace Bug Finder analysis on the example file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Enable checking of MISRA C:2012 rules. Check for the mandatory rules only.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = 'mandatory';

% Run analysis
bfStatus = run(proj, 'bugFinder');

% Read results
defectsSummary = proj.Results.getSummary('defects');
misraSummary = proj.Results.getSummary('misraC2012');
```

See Also

Topics

“Run Polyspace Analysis by Using MATLAB Scripts”
“Generate MATLAB Scripts from Polyspace User Interface”
“Troubleshoot Polyspace Analysis from MATLAB”

Introduced in R2017b

polyspace.Options class

Package: `polyspace`

Create object for running Polyspace analysis on handwritten code

Note For easier scripting, specify the Polyspace® analysis options using the `Configuration` property of a `polyspace.Project` object. Do not create a `polyspace.Options` object directly.

Description

Run a Polyspace analysis from MATLAB by using an options object. To specify source files and customize analysis options, change the object properties.

To analyze model-generated code (using the Polyspace desktop products), use `polyspace.ModelLinkOptions` instead.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

Construction

`opts = polyspace.Options` creates an object whose properties correspond to options for running a Polyspace analysis.

`proj = polyspace.Project` creates a `polyspace.Project` object. The object has a property `Configuration`, which is a `polyspace.Options` object.

`opts = polyspace.Options(lang)` creates a Polyspace options object with options that are applicable to the language `lang`.

`opts = polyspace.loadProject(projectFile)` creates a Polyspace options object from an existing Polyspace project `projectFile`. You set the options in your project in the Polyspace user interface and create the options object from that project for programmatically running the analysis.

Input Arguments

`lang` — Language of analysis

`'C-CPP'` (default) | `'C'` | `'CPP'`

The language of the analysis specified as `'C-CPP'`, `'C'`, or `'CPP'`. This argument determines the object properties.

Data Types: `char`

`projectFile` — Name of `.psprj` file

character vector

Name of Polyspace project file with extension `.psprj`, specified as a character vector.

If the file is not in the current folder, `projectFile` must include a full or relative path. To identify the current folder, use `pwd`. To change the current folder, use `cd`.

Example: `'C:\projects\myProject.psprj'`

Properties

The object properties correspond to the analysis options for Polyspace projects. The properties are organized in the same categories as the Polyspace interface. The property names are a shortened version of the DOS/UNIX command-line name. For syntax details, see polyspace.Project.Configuration properties.

Methods

copyTo	Copy common settings between Polyspace options objects
generateProject	Generate psprj project from options object
toScript	Add Polyspace options object definition to a script

Examples

Customize and Run Analysis

Create a Polyspace analysis options object and customize the properties. Then, run an analysis.

Create object and customize properties. In case you do not have write access to your current folder, a temporary folder is being used for storing analysis results.

```
sources = fullfile(polyspaceroot, 'polyspace','examples','cxx','Bug_Finder_Example',...
    'sources','numerical.c');
opts = polyspace.Options();
opts.Prog = 'MyProject';
opts.Sources = {sources};
opts.TargetCompiler.Compiler = 'gnu4.7';
opts.ResultsDir = tempname;
```

Run a Bug Finder analysis. To run a Code Prover analysis, use `polyspaceCodeProver` instead of `polyspaceBugFinder`.

```
results = polyspaceBugFinder(opts);
```

With the Polyspace Server products, you can use the functions `polyspaceBugFinderServer` or `polyspaceCodeProverServer`.

Open the results in the Polyspace user interface of the desktop products.

```
polyspaceBugFinder('-results-dir',opts.ResultsDir);
```

Run Polyspace by Generating a Project File

Create a Polyspace analysis options object and customize the properties. Then, run a Bug Finder analysis.

Create object and customize properties.

```
sources=fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example',...
    'sources','numerical.c');
opts = polyspace.Options();
opts.Prog = 'MyProject';
opts.Sources = {sources};
opts.TargetCompiler.Compiler = 'gnu4.7';
opts.ResultsDir = tempname;
```

Generate a Polyspace project, name it using the `Prog` property, and open the project in the Polyspace interface.

```
psprj = opts.generateProject(opts.Prog);
polyspaceBugFinder(psprj);
```

You can also analyze the project from the command line. Run the analysis and open the results in the Polyspace interface.

```
results = polyspaceBugFinder(psprj, '-nodesktop');
polyspaceBugFinder('-results-dir',opts.ResultsDir);
```

Alternatives

If you are analyzing code generated from a model, use `polyspace.ModelLinkOptions` instead.

See Also

`polyspace.ModelLinkOptions` | `polyspace.Project` | `polyspaceBugFinder`

Topics

“Run Polyspace Analysis by Using MATLAB Scripts”
“Generate MATLAB Scripts from Polyspace User Interface”

Introduced in R2017a

polyspace.ModelLinkOptions class

Package: `polyspace`

Create object for running Polyspace analysis on generated code

Description

Run a Polyspace analysis from MATLAB by using an options object. To specify source files and customize analysis options, change the object properties.

This class is intended for model-generated code. If you are analyzing handwritten code, use `polyspace.Options` instead.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

Construction

`opts = polyspace.ModelLinkOptions` creates an object whose properties correspond to options for running a Polyspace analysis on generated code.

`opts = polyspace.ModelLinkOptions(lang)` creates a Polyspace options object with options that are applicable to the language `lang`.

`opts = polyspace.ModelLinkOptions(model)` creates a Polyspace options object with options that are applicable to `model`. Prior to extracting options from the model, you must load the model and generate code.

Input Arguments

`lang` — Language of analysis

`'C-CPP'` (default) | `'C'` | `'CPP'`

The language of the analysis specified as `'C-CPP'`, `'C'`, or `'CPP'`. This argument determines the object properties.

`model` — Model or subsystem name

character vector

Name or path to model or subsystem, specified as a character vector.

Prior to extracting options from the model, you must:

1. Load the model. Use `load_system` or `open_system`.
2. Generate code from the model. Use `rtwbuild`.

Example: `'psdemo_model_link_sl'`

Properties

The object properties correspond to the analysis options for Polyspace projects. The properties are organized in the same categories as the Polyspace interface. The property names are a shortened version of the DOS command-line name. For syntax details, see polyspace.ModelLinkOptions.

Methods

copyTo	Copy common settings between Polyspace options objects
generateProject	Generate psprj project from options object
toScript	Add Polyspace options object definition to a script

Examples

Script Analysis of Model Generated Code

This example shows how to customize and run an analysis on code generated from a model.

Generate code from the model `sldemo_bounce`. Before code generation, set a system target file appropriate for code analysis. See also “Recommended Model Configuration Parameters for Polyspace Analysis”.

```
modelName = 'rtwdemo_roll';
load_system(modelName);

% Set parameters for Embedded Coder target
set_param(modelName, 'SystemTargetFile', 'ert.tlc');
set_param(modelName,'Solver','FixedStepDiscrete');
set_param(modelName,'SupportContinuousTime','on');
set_param(modelName,'LaunchReport','off');
set_param(modelName,'InitFltsAndDblsToZero','on');

if exist(fullfile(pwd,'rtwdemo_roll_ert_rtw'), 'dir') == 0
    rtwbuild(modelName);
end
```

Associate a `polyspace.ModelLinkOptions` object with the model. A subset of the object properties are set from the configuration parameters associated with the model. The other properties take their default values. For details on the configuration parameters, see “Polyspace Analysis in Simulink”.

```
opts = polyspace.ModelLinkOptions(modelName);
```

Change the property values if needed. For instance, you can specify that the analysis must check for all MISRA C: 2012 violations and generate a PDF report of the results. You can also specify a folder for the analysis results.

```
opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
opts.CodingRulesCodeMetrics.MisraC3Subset = 'all';

opts.MergedReporting.EnableReportGeneration = true;
opts.MergedReporting.ReportOutputFormat = 'PDF';

opts.ResultsDir = 'newResfolder';
```

Create a `polyspace.Project` object. Associate the `Configuration` property of this object to the options that you previously specified.

```
proj = polyspace.Project;
proj.Configuration = opts;
```

Run analysis and open results.

```
cpStatus = proj.run('codeProver');
proj.Results.getResults('readable');
```

Alternatives

If you are analyzing handwritten code, use a `polyspace.Projectpolyspace.Project` object directly. Alternatively, use a `polyspace.Options` object.

See Also

`polyspace.Options` | `polyspace.Project` | `polyspaceBugFinder` | `pslinkrun`

Topics

“Run Polyspace Analysis by Using MATLAB Scripts”

Introduced in R2017a

polyspace.BugFinderOptions class

Package: `polyspace`

Create Polyspace Bug Finder object for handwritten code

Note This class is deprecated and will be removed in a future release. Use `polyspace.Options` instead.

Description

Customize a Polyspace Bug Finder analysis from MATLAB by creating a Bug Finder options object. To specify source files and customize analysis options, change the object properties.

If you are analyzing model-generated code, use `polyspace.ModelLinkBugFinderOptions` instead.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

Construction

`opts = polyspace.BugFinderOptions` creates a Bug Finder options object with available options.

`opts = polyspace.BugFinderOptions(lang)` creates a Bug Finder options object with options that are applicable for the language `lang`.

Input Arguments

`lang` — Language of analysis
`'C-CPP'` (default) | `'C'` | `'CPP'`

The language of the analysis specified as `'C-CPP'`, `'C'`, or `'CPP'`. This argument determines which properties the object has.

Properties

The object properties are the analysis options for Polyspace Bug Finder projects. The properties are organized in the same categories as the Polyspace interface. The property names are a shortened version of the DOS/UNIX command-line name. For syntax details, see polyspace.Options.

Methods

copyTo	Copy common settings between Polyspace options objects
generateProject	Generate psprj project from options object
toScript	Add Polyspace options object definition to a script

Examples

Customize and Run Analysis

Create a Bug Finder analysis options object and customize the properties. Then, run an analysis.

Create object and customize properties.

```
sources = fullfile(polyspaceroot, 'polyspace','examples','cxx','Bug_Finder_Example',...
    'sources','numerical.c');
optsBF = polyspace.BugFinderOptions();
optsBF.Prog = 'MyProject';
optsBF.Sources = {sources};
optsBF.TargetCompiler.Compiler = 'gnu4.7';
optsBF.ResultsDir = tempname;
```

Run the analysis and open the results in the Polyspace interface.

```
results = polyspaceBugFinder(optsBF);
polyspaceBugFinder('-results-dir',optsBF.ResultsDir);
```

Run Polyspace by Generating a Project File

Create a Bug Finder analysis options object and customize the properties. Then, run an analysis.

Create object and customize properties.

```
sources = fullfile(polyspaceroot, 'polyspace','examples','cxx','Bug_Finder_Example',...
    'sources','numerical.c');
optsBF = polyspace.BugFinderOptions();
optsBF.Prog = 'MyProject';
optsBF.Sources = {sources};
optsBF.TargetCompiler.Compiler = 'gnu4.7';
optsBF.ResultsDir = tempname;
```

Generate a Polyspace project, name it using the `Prog` property, and open the project in the Polyspace interface.

```
psprj = generateProject(optsBF, optsBF.Prog);
polyspaceBugFinder(psprj);
```

Run the analysis and open the results in the Polyspace interface.

```
results = polyspaceBugFinder(psprj, '-nodesktop');
polyspaceBugFinder('-results-dir',optsBF.ResultsDir);
```

Alternatives

If you are analyzing code generated from a model, use `polyspace.ModelLinkBugFinderOptions` instead.

See Also

`polyspace.ModelLinkBugFinderOptions` | polyspace.Options | `polyspaceBugFinder`

Topics

“Run Polyspace Analysis by Using MATLAB Scripts”

Introduced in R2016b

polyspace.DefectsOptions class

Package: `polyspace`

Create custom list of defects to check

Description

Create a custom list of defects to check in a Polyspace analysis.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

Construction

`defectsList = polyspace.DefectsOptions` creates the defect options object `defectsList`. You can customize the list of active defects by changing the properties.

Properties

An object is created with supported defects as properties. The defects are listed by their command-line name. See “Short Names of Bug Finder Defect Checkers”.

By default, all defects are turned off. To turn on a defect, set the defect to true. For example:

```
defectsList = polyspace.DefectsOptions;
defectsList.FLOAT_ZERO_DIV = true;
```

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Customize List of Defects to Check

Customize the list of defects checked during a Polyspace Bug Finder analysis.

Create two objects: a `polyspace.DefectsOptions` object for setting coding rules and a `polyspace.Project` object for running the Polyspace analysis.

```
defectsList = polyspace.DefectsOptions;
proj = polyspace.Project;
```

Enable the numerical defects.

```
defectsList.FLOAT_ZERO_DIV = true;
defectsList.INT_ZERO_DIV = true;
defectsList.FLOAT_ABSORPTION = true;
```

```
defectsList.BITWISE_NEG = true;
defectsList.FLOAT_CONV_OVFL = true;
defectsList.FLOAT_OVFL = true;
defectsList.INT_CONV_OVFL = true;
defectsList.INT_OVFL = true;
defectsList.FLOAT_STD_LIB = true;
defectsList.INT_STD_LIB = true;
defectsList.SHIFT_NEG = true;
defectsList.SHIFT_OVFL = true;
defectsList.SIGN_CHANGE = true;
defectsList.UINT_CONV_OVFL = true;
defectsList.UINT_OVFL = true;
defectsList.BAD_PLAIN_CHAR_USE = true;
```

Add the customized list of defects to the `Configuration` property of the `polyspace.Project` object.

```
proj.Configuration.BugFinderAnalysis.CheckersList = defectsList;
proj.Configuration.BugFinderAnalysis.CheckersPreset = 'custom';
```

You can now use the `polyspace.Project` object to run the analysis.

See Also

`polyspace.CodingRulesOptions` | `polyspace.ModelLinkOptions` | `polyspace.Options` | `polyspace.Project`

Topics

“Short Names of Bug Finder Defect Checkers”

Introduced in R2016b

polyspace.ModelLinkBugFinderOptions class

Package: `polyspace`

Create Polyspace Bug Finder object for generated code

Note This class is deprecated and will be removed in a future release. Use `polyspace.ModelLinkOptions` instead.

Description

Customize a Polyspace Bug Finder analysis from MATLAB by creating a Bug Finder options object. To specify source files and customize analysis options, change the object properties.

This class is intended for model-generated code. If you are analyzing handwritten code, use `polyspace.BugFinderOptions` instead.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

Construction

`opts = polyspace.BugFinderOptions` creates a Bug Finder options object for generated code with available options for C/C++ generated code.

Properties

The object properties are the analysis options for Polyspace Bug Finder model link projects. The properties are organized in the same categories as the Polyspace interface. The property names are a shortened version of the DOS command-line name. For syntax details, see polyspace.ModelLinkOptions.

Methods

copyTo	Copy common settings between Polyspace options objects
generateProject	Generate psprj project from options object
toScript	Add Polyspace options object definition to a script

Examples

Script Analysis of Model Generated Code

This example shows how to customize and run an analysis on model generated code with MATLAB functions and objects.

Create a custom configuration that checks MISRA C 2012 rules and generates a PDF report.

```
opts = polyspace.ModelLinkBugFinderOptions();
opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
opts.CodingRulesCodeMetrics.MisraC3Subset = 'all';
opts.MergedReporting.ReportOutputFormat = 'PDF';
opts.MergedReporting.EnableReportGeneration = true;
```

Generate code from `psdemo_model_link_sl`.

```
model = 'psdemo_model_link_sl';
load_system(model);
slbuild(model);
```

Add the configuration to `pslinkoptions` object.

```
prjfile = opts.generateProject('model_link_opts');
mlopts = pslinkoptions(model);
mlopts.EnablePrjConfigFile = true;
mlopts.PrjConfigFile = prjfile;
mlopts.VerificationMode = 'BugFinder';
```

Run analysis.

```
[polyspaceFolder, resultsFolder] = pslinkrun(model);
```

Alternatives

If you are analyzing handwritten code, use `polyspace.BugFinderOptions` instead.

See Also

`polyspace.BugFinderOptions` | polyspace.ModelLinkOptions | `polyspaceBugFinder` | `pslinkrun`

Topics

“Run Polyspace Analysis by Using MATLAB Scripts”

polyspace.GenericTargetOptions class

Package: `polyspace`

Create a generic target configuration

Description

Create a custom target for a Polyspace analysis if your target processor does not match one of the predefined targets,.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

Construction

`genericTarget = polyspace.GenericTargetOptions` creates a generic target that you can customize. To specify the sizes and alignment of data types, change the properties of the object. For instance:

```
target = polyspace.GenericTargetOptions;
target.CharNumBits = 16;
```

Properties

For more details about any of the properties below, see `Generic target options`.

`Alignment` — Largest alignment of struct or array objects
32 (default) | 16 | 8

Largest alignment of struct or array objects, specified as `32`, `16`, or `8`. Comparable with the DOS/UNIX command-line option `-align`.

Example: `target.Alignment = 8`

`CharNumBits` — Define the number of bits for a `char`
8 (default) | 16

Define the number of bits for a `char`, specified as `8` or `16`. Comparable with the DOS/UNIX command-line option `-char-is-16bits`.

Example: `target.CharNumBits = 16`

`DoubleNumBits` — Define the number of bits for a `double`
32 (default) | 64

Define the number of bits for a `double`, specified as `32` or `64`. Comparable with the DOS/UNIX command-line option `-double-is-64bits`.

Example: `target.DoubleNumBits = 64`

`Endianness` — Endianness of target architecture
`little` (default) | `big`

Endianness of target architecture, specified as `little` or `big`. Comparable with the DOS/UNIX command-line options `-little-endian` or `-big-endian`.

Example: `target.Endianess = 'big'`

`IntNumBits` — Define the number of bits for an `int`
16 (default) | 32

Define the number of bits for an `int`, specified as 16 or 32. Comparable with the DOS/UNIX command-line option `-int-is-32bits`.

Example: `target.IntNumBits = 32`

`LongLongNumBits` — Define the number of bits for a `long long`
32 (default) | 64

Define the number of bits for a `long long`, specified as 32 or 64. Comparable with the DOS/UNIX command-line option `-long-long-is-64bits`.

Example: `target.LongNumBits = 64`

`LongNumBits` — Define the number of bits for a `long`
32 (default)

Define the number of bits for a `long`, specified as 32. Comparable with the DOS/UNIX command-line option `-long-is-32bits`.

Example: `target.LongNumBits = 32`

`PointerNumBits` — Define the number of bits for a pointer
16 (default) | 24 | 32

Define the number of bits for a pointer, specified as 16, 24, or 32. Comparable with the DOS/UNIX command-line options `-pointer-is-24bits` and `-pointer-is-32bits`.

Example: `target.PointerNumBits = 32`

`ShortNumBits` — Define the number of bits for a `short`
16 (default) | 8

Define the number of bits for an `int`, specified as 16 or 8. Comparable with the DOS/UNIX command-line option `-short-is-8bits`.

Example: `target.ShortNumBits = 8`

`SignOfChar` — Default sign of plain `char`
`signed` (default) | `unsigned`

Default sign of plain `char`, specified as `signed` or `unsigned`. Comparable with the DOS/UNIX command-line option `-default-sign-of-char`.

Example: `target.SignOfChar = 'unsigned'`

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Customize Generic Target Settings

Use a custom target for the Polyspace analysis.

Create two objects: a `polyspace.GenericTargetOptions` object for creating a custom target and a `polyspace.Project` object for running the Polyspace analysis.

```
target = polyspace.GenericTargetOptions;
proj = polyspace.Project;
```

Customize the generic target.

```
target.Endianess = 'big';
target.LongLongNumBits = 64;
target.ShortNumBits = 8;
```

Add the custom target to the `Configuration` property of the `polyspace.Project` object.

```
proj.Configuration.TargetCompiler.Target = target;
```

You can now use the `polyspace.Project` object to run the analysis.

`Generic target options` | `polyspace.CodingRulesOptions` | `polyspace.ModelLinkOptions` | `polyspace.Options` | `polyspace.Project`

Introduced in R2016b

polyspace.CodingRulesOptions class

Package: `polyspace`

Create custom list of coding rules to check

Description

Create a custom list of coding rules to check in a Polyspace analysis.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

Construction

`ruleList = polyspace.CodingRulesOptions(RuleSet)` creates the coding rules object `ruleList` for the `RuleSet` coding rule set. Set the active rules in the coding rules object.

Input Arguments

`RuleSet` — Standard coding rule set

`misraC` (default) | `misraC2012` | `misraAcAgc` | `misraCpp` | `jsf` | `certC` | `certCpp` | `iso17961` | `autosarCpp14`

Standard coding rule set specified as one of the coding rule acronyms.

Example: `'misraCpp'`

Data Types: `char`

Properties

For each coding rule set, an object is created with all supported rules divided into sections. By default, all rules are on. To turn off a rule, set the rule to false. For example:

```
misraRules = polyspace.CodingRulesOptions('misraC');
misraRules.Section_20_Standard_libraries.rule_20_1 = false;
```

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Customize List of Coding Rules to Check

Customize the coding rules that are checked in a Polyspace analysis. Since all rules are enabled by default, you can create a custom subset by disabling some rules.

Create two objects: a `polyspace.CodingRulesOptions` object for setting coding rules and a `polyspace.Project` object for running the Polyspace analysis.

```
misraRules = polyspace.CodingRulesOptions('misraC2012');
proj = polyspace.Project;
```

Customize the coding rule list by turning off rules 2.1-2.7.

```
misraRules.Section_2_Unused_code.rule_2_1 = false;
misraRules.Section_2_Unused_code.rule_2_2 = false;
misraRules.Section_2_Unused_code.rule_2_3 = false;
misraRules.Section_2_Unused_code.rule_2_4 = false;
misraRules.Section_2_Unused_code.rule_2_5 = false;
misraRules.Section_2_Unused_code.rule_2_6 = false;
misraRules.Section_2_Unused_code.rule_2_7 = false;
```

Add the customized list of coding rules to the `Configuration` property of the `polyspace.Project` object.

```
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = misraRules;
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.CodingRulesCodeMetrics.EnableCheckersSelectionByFile = true;
```

You have to enable checkers selection by file because the Polyspace run uses an XML file underneath to enable the coding rule checkers. The XML file is saved in a `.settings` subfolder of the results folder.

You can now use the `polyspace.Project` object to run the analysis. For instance, you can enter:

```
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
run(proj, 'bugfinder');
```

Create Coding Rules Object Using Rule Numbers to Enable

Suppose that you want to specify a subset of MISRA C: 2012 rules for the analysis. Instead of enumerating rules that you want disabled, you can specify the rules that you want to keep enabled. You can also specify the rule numbers only without the MISRA C: 2012 sections containing the rules.

Specify the rule numbers in a cell array to the `createRulesObject` function defined as follows.

```
function rulesObject = createRulesObject(rulesToEnable)

%% This function takes a cell array of MISRA C:2012 rules and returns
%% a polyspace.CodingRulesOptions object with the rules enabled.
%% Example input argument: {'2.7', '3.1'}

    rulesObject = polyspace.CodingRulesOptions('misraC2012');

    % Coding Standards documents have many sections. Loop over all
    % sections.
    ruleSections = properties(rulesObject);
    for i=1:length(ruleSections)
         sectionName = ruleSections{i};
         rulesInSection = properties(rulesObject.(sectionName));
```

```
            % Loop over all rules in a section, enable or disable rule based
            % on input
            for j=1:length(rulesInSection)
                ruleNumberAsProperty = rulesInSection{j};
                ruleNumber = strrep(strrep(ruleNumberAsProperty,'rule_',''),'_','.');
                if(any(strcmp(rulesToEnable,ruleNumber)))
                    rulesObject.(sectionName).(ruleNumberAsProperty)=1;
                else
                    rulesObject.(sectionName).(ruleNumberAsProperty)=0;
                end
            end
    end
end
```

For instance, to enable rules 1.1 and 2.2, enter:

```
createRulesObject({'1.1','2.2'})
```

See Also

`polyspace.ModelLinkOptions` | `polyspace.Options` | `polyspace.Project`

Introduced in R2016b

polyspace.BugFinderResults class

Package: `polyspace`

Read Polyspace Bug Finder results from MATLAB

Description

Read Polyspace Bug Finder analysis results to MATLAB tables by using this object.

You can obtain a high-level overview or read each individual result, for example, each instance of a defect.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

Construction

`resObj = polyspace.BugFinderResults(resultsFolder)` creates an object for reading a specific set of Bug Finder results into MATLAB tables. Use the object methods to read the results.

`proj = polyspace.Project` creates a `polyspace.Project` object. The object has a property `Results`. If you run a Bug Finder analysis, this property is a `polyspace.BugFinderResults` object.

Input Arguments

`resultsFolder` — Name of result folder
character vector

Name of result folder, specified as a character vector. The folder must contain the results file with extension `.psbf`. Even if the results file resides in a *subfolder* of the specified folder, it cannot be accessed.

If the folder is not in the current folder, `resultsFolder` must include a full or relative path.

Example: `'C:\Polyspace\Results\'`

Methods

getSummary	View number of defects organized by defect type
getResults	Read Bug Finder results into MATLAB table

Examples

Copy Existing Results to MATLAB Tables

This example shows how to read Bug Finder analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath=fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example',...
'Module_1','BF_Result');
userResPath = tempname;
copyfile(resPath,userResPath);
```

Create the results object.

```
resObj = polyspace.BugFinderResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = getSummary (resObj);
resTable = getResults (resObj);
```

Run Analysis and Read Results to MATLAB Tables

Run a Polyspace Bug Finder analysis on the demo file `numerical.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace',...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = run(proj, 'bugFinder');

% Read results
bfSummary = proj.Results.getResults('readable');
```

Alternatives

To read Code Prover results from MATLAB, use the class `polyspace.CodeProverResults`. See `polyspace.CodeProverResults`.

Introduced in R2017a

pslinkoptions Properties

Properties for the `pslinkoptions` object

Description

You can create a `pslinkoptions` object to customize your analysis at the command-line. Use these properties to specify configuration options, where and how to store results, additional files to include, and data range modes.

Properties

Configuration Options

`VerificationSettings` — Coding rule and configuration settings for C code
`'PrjConfig'` (default) | `'PrjConfigAndMisraAGC'` | `'PrjConfigAndMisra'` | `'PrjConfigAndMisraC2012'` | `'MisraAGC'` | `'Misra'` | `'MisraC2012'`

Coding rule and configuration settings for C code specified as:

- `'PrjConfig'` – Inherit options from the project configuration.
- `'PrjConfigAndMisraAGC'` – Inherit options from the project configuration and enable MISRA AC AGC rule checking.
- `'PrjConfigAndMisra'` – Inherit options from the project configuration and enable MISRA C:2004 rule checking.
- `'PrjConfigAndMisraC2012'` – Inherit options from the project configuration and enable MISRA C:2012 guideline checking.
- `'MisraAGC'` – Enable MISRA AC AGC rule checking. This option runs only compilation and rule checking.
- `'Misra'` – Enable MISRA C:2004 rule checking. This option runs only compilation and rule checking.
- `'MisraC2012'` – Enable MISRA C:2012 rule checking. This option runs only compilation and guideline checking.

Example: `opt.VerificationSettings = 'PrjConfigAndMisraC2012'`

`VerificationMode` — Polyspace mode
`'BugFinder'` (default) | `'CodeProver'`

Polyspace mode specified as `'BugFinder'`, for a Bug Finder analysis, or `'CodeProver'`, for a Code Prover verification.

Example: `opt.VerificationMode = 'BugFinder';`

`EnablePrjConfigFile` — Allow a custom configuration file
`false` (default) | `true`

Allows a custom configuration file instead of the default configuration specified as true or false. Use the `PrjConfigFile` option to specify the configuration file.

Example: `opt.EnablePrjConfigFile = true;`

`PrjConfigFile` — Custom configuration file
`''` (default) | full path to a `.psprj` file

Custom configuration file to use instead of the default configuration specified by the full path to a `.psprj` file. Use the `EnablePrjConfigFile` option to use this configuration file during your analysis.

Example: `opt.PrjConfigFile = 'C:\Polyspace\config.psprj';`

`CheckConfigBeforeAnalysis` — Configuration check before analysis
`'OnWarn'` (default) | `'OnHalt'` | `'Off'`

This property sets the level of configuration checking done before the analysis starts. The configuration check before analysis is specified as:

- **`'Off'`** — Checks only for errors. Stops if errors are found.
- **`'OnWarn'`** — Stops for errors. Displays a message for warnings.
- **`'OnHalt'`** — Stops for errors and warnings.

Example: `opt.CheckConfigBeforeAnalysis = 'OnHalt';`

Results

`ResultDir` — Results folder name and location
`'C:\Polyspace_Results\results_$ModelName$'` (default) | folder name | folder path

Results folder name and location specified as the local folder name or the folder path. This folder is where Polyspace writes the analysis results. This folder name can be either an absolute path or a path relative to the current folder. The text `$ModelName$` is replaced with the name of the original model.

Example: `opt.ResultDir = '\results_v1_$ModelName$';`

`AddSuffixToResultDir` — Add unique number to the results folder name
`false` (default) | `true`

Add unique number to the results folder name specified as true or false. If true, a unique number is added to the end of every new result. Using this option helps you avoid overwriting the previous results folders.

Example: `opt.AddSuffixToResultDir = true;`

`OpenProjectManager` — Open the Polyspace environment
`false` (default) | `true`

Open the Polyspace environment to monitor the progress of the analysis, specified as true or false. Afterward, you can review the results.

Example: `opt.OpenProjectManager = true;`

`AddToSimulinkProject` — Add results to the open Simulink project
`false` (default) | `true`

Add your results to the currently open Simulink project, if any, specified as true or false. This option allows you to keep your Polyspace results organized with the rest of your project files. If a Simulink project is not open, the results are not added to a Simulink project.

Example: `opt.AddToSimulinkProject = true;`

Additional Files

`EnableAdditionalFileList` — Allow an additional file list
`false` (default) | `true`

Allow an additional file list to be analyzed, specified as true or false. Use with the `AdditionalFileList` option.

Example: `opt.EnableAdditionalFileList = true;`

`AdditionalFileList` — List of additional files to be analyzed
`{0x1 cell}` (default) | cell array of files

List of additional files to be analyzed specified as a cell array of files. Use with the `EnableAdditionalFileList` option to add these files to the analysis.

Example: `opt.AdditionalFileList = {'sources\file1.c', 'sources\file2.c'};`

Data Types: `cell`

Data Ranges

`InputRangeMode` — Enable design range information
`'DesignMinMax'` (default) | `'FullRange'`

Enable design range information specified as `'DesignMinMax'`, to use data ranges defined in blocks and workspaces, or `'FullRange'`, to treat inputs as full-range values.

Example: `opt.InputRangeMode = 'FullRange';`

`ParamRangeMode` — Enable constant parameter values
`'None'` (default) | `'DesignMinMax'`

Enable constant parameter values, specified as `'None'`, to use constant parameters values specified in the code, or `'DesignMinMax'` to use a range defined in blocks and workspaces.

Example: `opt.ParamRangeMode = 'DesignMinMax';`

`OutputRangeMode` — Enable output assertions
`'None'` (default) | `'DesignMinMax'`

Enable output assertions specified by `'None'`, to not apply assertions, or `'DesignMinMax'` to apply assertions to outputs using a range defined in blocks and workspace.

Example: `opt.ParamRangeMode = 'DesignMinMax';`

Embedded Coder Only

`ModelRefVerifDepth` — Depth of verification
`'Current model only'` (default) | `'1'` | `'2'` | `'3'` | `'All'`

Depth of verification specified by the model reference level to which you want to analyze.

Only for Embedded Coder

Example: `opt.ModelRefVerifDepth = '3';`

`ModelRefByModelRefVerif` — Model reference analysis mode
`false` (default) | `true`

Model reference analysis mode specified as `false` to verify reference models within the model hierarchy, or `true` to verify referenced models individually.

Only for Embedded Coder

Example: `opt.ModelRefByModelRefVerif = true;`

`CxxVerificationSettings` — Coding rule and configuration settings for C++ code
`'PrjConfig'` (default) | `'PrjConfigAndMisraCxx'` | `'PrjConfigAndJSF'` | `'MisraCxx'` | `'JSF'`

Coding rule and configuration settings for C++ code specified as:

- `'PrjConfig'` – Inherit options from project configuration and run complete analysis.
- `'PrjConfigAndMisraCxx'` – Inherit options from project configuration, enable MISRA C++ rule checking, and run complete analysis.
- `'PrjConfigAndJSF'` – Inherit options from project configuration, enable JSF rule checking, and run complete analysis.
- `'MisraCxx'` – Enable MISRA C++ rule checking, and run compilation phase only.
- `'JSF'` – Enable JSF rule checking, and run compilation phase only.

Only for Embedded Coder

Example: `opt.CxxVerificationSettings = 'MisraCxx';`

TargetLink Only

`AutoStubLUT` — Lookup Table code usage
`false` (default) | `true`

Lookup Table code usage, specified as true or false.

- `true` — use Lookup Table code during the analysis.
- `false` — stub Lookup Table code.

Only for TargetLink

Example: `opts.AutoStubLUT = true;`

See Also

`pslinkoptions` | `pslinkrun`

polyspace.Project.Configuration Properties

Customize Polyspace analysis of handwritten code with options object properties

Description

To customize your Polyspace analysis, use these `polyspace.Options` or `polyspace.Project.Configuration` properties. Each property corresponds to an analysis option on the **Configuration** pane in the Polyspace user interface.

The properties are grouped using the same categories as the **Configuration** pane. This page only shows what values each property can take. For details about:

- The different options, see the analysis option reference pages.
- How to create and use the object, see `polyspace.Options` or `polyspace.Project`.

  The same properties are also available with the deprecated classes `polyspace.BugFinderOptions` and `polyspace.CodeProverOptions`.

Each property description below also highlights if the option affects only one of Bug Finder or Code Prover.

Note Some options might not be available depending on the language setting of the object. You can set the source code language (`Language`) to `'C'`, `'CPP'` or `'C-CPP'` during object creation, but cannot change it later.

Properties

Advanced

`Additional` — Additional flags for analysis

character vector

Additional flags for analysis specified as a character vector.

For more information, see `Other`.

Example: `opts.Advanced.Additional = '-extra-flags -option -extra-flags value'`

`PostAnalysisCommand` — Command or script software should execute after analysis finishes

character vector

Command or script software should execute after analysis finishes, specified as a character vector.

For more information, see `Command/script to apply after the end of the code verification (-post-analysis-command)`.

Example: `opts.Advanced.PostAnalysisCommand = '"C:\Program Files\perl\win32\bin\perl.exe" "C:\My_Scripts\send_email"'`

`AutomaticOrangeTester` — Run the Automatic Orange Tester
false (default) | true

This property affects Code Prover analysis only.

Run the Automatic Orange Tester after verification, specified as true or false.

For more information, see `Automatic Orange Tester (-automatic-orange-tester)`.

Example: `opts.Advanced.AutomaticOrangeTester = true`

`AutomaticOrangeTesterLoopMaxIteration` — Number of loop iterations after which Automatic Orange Tester considers infinite loop
1000 (default) | positive integer

This property affects Code Prover analysis only.

Number of loop iterations after which Automatic Orange Tester considers the test an infinite loop, specified as a positive integer, maximum of 1000.

For more information, see `Maximum loop iterations (-automatic-orange-tester-loop-max-iteration)`.

Example: `opts.Advanced.AutomaticOrangeTesterLoopMaxIteration = 500`

`AutomaticOrangeTesterTestsNumber` — Number of tests that Automatic Orange Tester must run
500 (default) | positive integer

This property affects Code Prover analysis only.

Number of tests that Automatic Orange Tester must run, specified as a positive integer, maximum of 100,000.

For more information, see `Number of automatic tests (-automatic-orange-tester-tests-number)`.

Example: `opts.Advanced.AutomaticOrangeTesterTestsNumber = 1000`

`AutomaticOrangeTesterTimeout` — Time in seconds allowed for a single test in Automatic Orange Tester
5 (default) | positive integer

This property affects Code Prover analysis only.

Time in seconds allowed for a single test in Automatic Orange Tester, specified as a positive integer, maximum of 60.

For more information, see `Maximum test time (-automatic-orange-tester-timeout)`.

Example: `opts.Advanced.AutomaticOrangeTesterTimeout = 10`

BugFinderAnalysis (Affects Bug Finder Only)

`CheckersList` — List of custom checkers to activate
`polyspace.DefectsOptions` object | cell array of defect acronyms

This property affects Bug Finder analysis only.

List of custom checkers to activate specified by using the name of a `polyspace.DefectsOptions` object or a cell array of defect acronyms. To use this custom list in your analysis, set `CheckersPreset` to `custom`.

For more information, see `polyspace.DefectsOptions`.

Example: `defects = polyspace.DefectsOptions;`
`opts.BugFinderAnalysis.CheckersList = defects`

Example: `opts.BugFinderAnalysis.CheckersList = {'INT_ZERO_DIV','FLOAT_ZERO_DIV'}`

`CheckersPreset` — Subset of Bug Finder defects
`'default'` (default) | `'all'` | `'CWE'` | `'custom'`

This property affects Bug Finder analysis only.

Preset checker list, specified as a character vector of one of the preset options: `'default'`, `'all'`, `'CWE'`,or `'custom'`. To use `'custom'`, specify a value for the property `BugFinderAnalysis.CheckersList`.

For more information, see `Find defects (-checkers)`.

Example: `opts.BugFinderAnalysis.CheckersPreset = 'all'`

`ChecksUsingSystemInputValues` — Activate stricter checks for system inputs
false (default) | true

This property affects Bug Finder analysis only.

Activate stricter checks that consider all possible value for:

- Global variables.
- Reads of volatile variables.
- Returns of stubbed functions.
- Inputs to functions specified with **SystemInputsFrom**.

The analysis considers all possible values for a subset of **Numerical** and **Static memory** defects.

This property is equivalent to the **Run stricter checks considering all values of system inputs** check box in the Polyspace interface.

For more information, see `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`

Example: `opts.BugFinderAnalysis.ChecksUsingSystemInputValues = true`

`EnableCheckers` — Activate defect checking
true (default) | false

This property affects Bug Finder analysis only.

Activate defect checking, specified as true or false. Setting this property to false disables all defects. If you want to disable defect checking but still get results, turn on coding rules checking or code metric checking.

This property is equivalent to the **Find defects** check box in the Polyspace interface.

Example: `opts.BugFinderAnalysis.EnableCheckers = false`

`SystemInputsFrom` — List of functions for which you run stricter checks
`'auto'` (default) | `'uncalled'` | `'all'` | `'custom'`

This property affects Bug Finder analysis only.

Functions for which you want to run stricter checks that consider all possible values of the function inputs. Specify the list of functions as `'auto'`, `'uncalled'`, `'all'`, or as a character array beginning with `custom=` followed by a comma-separated list of function names.

To enable this option, set `BugFinderAnalysis.ChecksUsingSystemInputValues = true`.

For more information, see `Consider inputs to these functions (-system-inputs-from)`

Example: `opts.BugFinderAnalysis.SystemInputsFrom = 'custom=foo,bar'`

ChecksAssumption (Affects Code Prover Only)

`AllowNegativeOperandInShift` — Allow left shift operations on a negative number
false (default) | true

This property affects Code Prover analysis only.

Allow left shift operations on a negative number, specified as true or false.

For more information, see `Allow negative operand for left shifts (-allow-negative-operand-in-shift)`.

Example: `opts.ChecksAssumption.AllowNegativeOperandInShift = true`

`AllowNonFiniteFloats` — Incorporate infinities and/or NaNs
false (default) | true

This property affects Code Prover analysis only.

Incorporate infinities and/or NaNs, specified as true or false.

For more information, see `Consider non finite floats (-allow-non-finite-floats)`.

Example: `opts.ChecksAssumption.AllowNonFiniteFloats = true`

`AllowPtrArithOnStruct` — Allow arithmetic on pointer to a structure field so that it points to another field
false (default) | true

This property affects Code Prover analysis only.

Allow arithmetic on pointer to a structure field so that it points to another field, specified as true or false.

For more information, see `Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)`.

Example: `opts.ChecksAssumption.AllowPtrArithOnStruct = true`

`CheckInfinite` — Detect floating-point operations that result in infinities
`'allow'` (default) | `'warn-first'` | `'forbid'`

This property affects Code Prover analysis only.

Detect floating-point operations that result in infinities.

To activate this option, specify `ChecksAssumption.AllowNonFiniteFloats`.

For more information, see `Infinities (-check-infinite)`.

Example: `opts.ChecksAssumption.CheckInfinite = 'forbid'`

`CheckNan` — Detect floating-point operations that result in NaN-s
`'allow'` (default) | `'warn-first'` | `'forbid'`

This property affects Code Prover analysis only.

Detect floating-point operations that result in NaN-s.

To activate this option, specify `ChecksAssumption.AllowNonFiniteFloats`.

For more information, see `NaNs (-check-nan)`.

Example: `opts.ChecksAssumption.CheckNan = 'forbid'`

`CheckSubnormal` — Detect operations that result in subnormal floating point values
`'allow'` (default) | `'warn-first'` | `'warn-all'` | `'forbid'`

This property affects Code Prover analysis only.

Detect operations that result in subnormal floating point values.

For more information, see `Subnormal detection mode (-check-subnormal)`.

Example: `opts.ChecksAssumption.CheckSubnormal = 'forbid'`

`DetectPointerEscape` — Find cases where a function returns a pointer to one of its local variables
false (default) | true

This property affects Code Prover analysis only.

Find cases where a function returns a pointer to one of its local variables, specified as true or false.

For more information, see `Detect stack pointer dereference outside scope (-detect-pointer-escape)`.

Example: `opts.ChecksAssumption.DetectPointerEscape = true`

`DisableInitializationChecks` — Disable checks for noninitialized variables and pointers
false (default) | true

This property affects Code Prover analysis only.

Disable checks for noninitialized variables and pointers, specified as true or false.

For more information, see `Disable checks for non-initialization (-disable-initialization-checks)`.

Example: `opts.ChecksAssumption.DisableInitializationChecks = true`

`PermissiveFunctionPointer` — Allow type mismatch between function pointers and the functions they point to
false (default) | true

This property affects Code Prover analysis only.

Allow type mismatch between function pointers and the functions they point to, specified as true or false.

For more information, see `Permissive function pointer calls (-permissive-function-pointer)`.

Example: `opts.ChecksAssumption.PermissiveFunctionPointer = true`

`SignedIntegerOverflows` — Behavior of signed integer overflows
`'forbid'` (default) | `'allow'` | `'warn-with-wrap-around'`

This property affects Code Prover analysis only.

Enable the check for signed integer overflows and the assumptions to make following an overflow specified as `'forbid'`, `'allow'`, or `'warn-with-wrap-around'`.

For more information, see `Overflow mode for signed integer (-signed-integer-overflows)`.

Example: `opts.ChecksAssumption.SignedIntegerOverflows = 'warn-with-wrap-around'`

`SizeInBytes` — Allow a pointer with insufficient memory buffer to point to a structure
false (default) | true

This property affects Code Prover analysis only.

Allow a pointer with insufficient memory buffer to point to a structure, specified as true or false.

For more information, see `Allow incomplete or partial allocation of structures (-size-in-bytes)`.

Example: `opts.ChecksAssumption.SizeInBytes = true`

`UncalledFunctionCheck` — Detect functions that are not called directly or indirectly from main or another entry-point function
`'none'` (default) | `'never-called'` | `'called-from-unreachable'` | `'all'`

This property affects Code Prover analysis only.

Detect functions that are not called directly or indirectly from main or another entry-point function, specified as `none`, `never-called`, `called-from-unreachable`, or `all`.

For more information, see `Detect uncalled functions (-uncalled-function-checks)`.

Example: `opts.ChecksAssumption.UncalledFunctionCheck = 'all'`

`UnsignedIntegerOverflows` — Behavior of unsigned integer overflows
`'allow'` (default) | `'forbid'` | `'warn-with-wrap-around'`

This property affects Code Prover analysis only.

Enable the check for unsigned integer overflows and the assumptions to make following an overflow, specified as `'forbid'`, `'allow'`, or `'warn-with-wrap-around'`.

For more information, see `Overflow mode for unsigned integer (-unsigned-integer-overflows)`.

Example: `opts.ChecksAssumption.UnsignedIntegerOverflows = 'allow'`

CodeProverVerification (Affects Code Prover only)

`ClassAnalyzer` — Classes that you want to verify
`'all'` (default) | `'none'` | `'custom=`*`class1[,class2,...]`*`'`

This property affects Code Prover analysis only.

Classes that you want to verify, specified as `'all'`, `'none'`, or as a character array beginning with `custom=` followed by a comma-separated list of class names.

For more information, see `Class (-class-analyzer)`.

Example: `opts.CodeProverVerification.ClassAnalyzer = 'custom=myClass1,myClass2'`

`ClassAnalyzerCalls` — Class methods that you want to verify
`'unused'` (default) | `'all'` | `'all-public'` | `'inherited-all'` | `'inherited-all-public'` | `'unused-public'` | `'inherited-unused'` | `'inherited-unused-public'` | `'custom=`*`method1[,method2,...]`*`'`

This property affects Code Prover analysis only.

Class methods that you want to verify, specified as one of the predefined sets or as a character array beginning with `custom=` followed by a comma-separated list of method names.

For more information, see `Functions to call within the specified classes (-class-analyzer-calls)`.

Example: `opts.CodeProverVerification.ClassAnalyzerCalls = 'unused-public'`

`ClassOnly` — Analyze only class methods
false (default) | true

This property affects Code Prover analysis only.

Analyze only class methods, specified as true or false.

For more information, see `Analyze class contents only (-class-only)`.

Example: `opts.CodeProverVerification.ClassOnly = true`

`EnableMain` — Use `main` function provided in application
false (default) | true

This property affects Code Prover analysis only.

Use `main` function provided in application, specified as true or false. If you set this property to false, the analysis generates a `main` function, if it is not present in the source files.

For more information, see `Verify whole application`.

Example: `opts.CodeProverVerification.EnableMain = true`

`FunctionsCalledBeforeMain` — Functions that you want the generated main to call ahead of other functions
cell array of function names

This property affects Code Prover analysis only.

Functions that you want the generated main to call ahead of other functions, specified as a cell array of function names.

For more information, see `Initialization functions (-functions-called-before-main)`.

Example: `opts.CodeProverVerification.FunctionsCalledBeforeMain = {'func1','func2'}`

`Main` — Use a Microsoft Visual C++ extensions of main
`'_tmain'` (default) | `'wmain'` | `'_tWinMain'` | `'wWinMain'` | `'WinMain'` | `'DllMain'`

This property applies to a Code Prover analysis only .

Use a Microsoft Visual C++ extension of main, specified as one of the predefined main extensions.

For more information, see `Main entry point (-main)`.

Example: `opts.CodeProverVerification.Main = 'wmain'`

`MainGenerator` — Generate a main function if it is not present in source files
true (default) | false

This property applies to a Code Prover analysis only .

Generate a main function if it is not present in source files, specified as true or false.

For more information, see `Verify module or library (-main-generator)`.

Example: `opts.CodeProverVerification.MainGenerator = false`

`MainGeneratorCalls` — Functions that you want the generated main to call after the initialization functions
`'unused'` (default) | `'none'` | `'all'` | `'custom=`*`function1[,function2,...]`*`'`

This property applies to a Code Prover analysis only .

Functions that you want the generated main to call after the initialization functions, specified as `'unused'`, `'all'`, `'none'`, or as a character array beginning with `custom=` followed by a comma-separated list of function names.

For more information, see `Functions to call (-main-generator-calls)`.

Example: `opts.CodeProverVerification.MainGeneratorCalls = 'all'`

`MainGeneratorWriteVariables` — Global variables that you want the generated main to initialize
`'uninit'` (C++ default) | `'public'` (C default) | `'none'` | `'all'` | `'custom=variable1[,variable2,...]'`

This property applies to a Code Prover analysis only .

Global variables that you want the generated main to initialize, specified as one of the predefined sets, or as a character array beginning with `custom=` followed by a comma-separated list of variable names.

For more information, see `Variables to initialize (-main-generator-writes-variables)`.

Example: `opts.CodeProverVerification.MainGeneratorWriteVariables = 'all'`

`NoConstructorsInitCheck` — Do not check if class constructor initializes class members
false (default) | true

This property applies to a Code Prover analysis only .

Do not check if class constructor initializes class members, specified as true or false.

For more information, see `Skip member initialization check (-no-constructors-init-check)`.

Example: `opts.CodeProverVerification.NoConstructorsInitCheck = true`

`UnitByUnit` — Verify each source file independently of other source files
false (default) | true

This property affects Code Prover analysis only.

Verify each source file independently of other source files, specified as true or false.

For more information, see `Verify files independently (-unit-by-unit)`.

Example: `opts.CodeProverVerification.UnitByUnit = true`

`UnitByUnitCommonSource` — Files that you want to include with each source file during a file-by-file verification
cell array of file paths

This property affects Code Prover analysis only.

Files that you want to include with each source file during a file-by-file verification, specified as a cell array of file paths.

For more information, see `Common source files (-unit-by-unit-common-source)`.

Example: `opts.CodeProverVerification.UnitByUnitCommonSource = {'/inc/file1.h','/inc/file2.h'}`

CodingRulesCodeMetrics

`AcAgcSubset` — Subset of MISRA AC AGC rules to check
`'OBL-rules'` (default) | `'OBL-REC-rules'` | `'single-unit-rules'` | `'system-decidable-rules'` | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA AC AGC rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA AC AGC (-misra-ac-agc)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA AC AGC rules, also set `EnableAcAgc` to true.

Example: `opts.CodingRulesCodeMetrics.AcAgcSubset = 'all-rules'`

Data Types: `char`

`AllowedPragmas` — Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied
cell array of character vectors

Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied, specified as a cell array of character vectors. This property affects only MISRA C:2004 or MISRA AC AGC rule checking.

For more information, see `Allowed pragmas (-allowed-pragmas)`.

Example: `opts.CodingRulesCodeMetrics.AllowedPragmas = {'pragma_01','pragma_02'}`

Data Types: `cell`

`AutosarCpp14` — Set of AUTOSAR C++ 14 rules to check
`'all'` (default) | `'required'` | `'automated'` | `polyspace.CodingRulesOptions` object | `'from-file'`

This property affects Bug Finder only.

Set of AUTOSAR C++ 14 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check AUTOSAR C++ 14 security checks (-autosar-cpp14)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.

- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check AUTOSAR C++ 14 rules, also set `EnableAutosarCpp14` to true.

Example: `opts.CodingRulesCodeMetrics.AutosarCpp14 = 'all'`

Data Types: `char`

`BooleanTypes` — Data types the coding rule checker must treat as effectively Boolean
cell array of character vectors

Data types that the coding rule checker must treat as effectively Boolean, specified as a cell array of character vectors.

For more information, see `Effective boolean types (-boolean-types)`.

Example: `opts.CodingRulesCodeMetrics.BooleanTypes = {'boolean1_t','boolean2_t'}`

Data Types: `cell`

`CertC` — Set of CERT C rules and recommendations to check
`'all'` (default) | `'publish-2016'` | `'all-rules'` | `polyspace.CodingRulesOptions` object | `'from-file'`

This property affects Bug Finder only.

Set of CERT C rules and recommendations to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check CERT-C security checks (-cert-c)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `from-file` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check CERT C rules and recommendations, also set `EnableCertC` to true.

Example: `opts.CodingRulesCodeMetrics.CertC = 'all'`

Data Types: `char`

`CertCpp` — Set of CERT C++ rules to check
`'all'` (default) | `polyspace.CodingRulesOptions` object | `'from-file'`

This property affects Bug Finder only.

Set of CERT C++ rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check CERT-C++ security checks (-cert-cpp)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check CERT C++ rules, also set `EnableCertCpp` to true.

Example: `opts.CodingRulesCodeMetrics.CertCpp = 'all'`

Data Types: `char`

`CheckersSelectionByFile` — File that defines custom set of coding standard checkers
full file path of `.xml` file

File where you define a custom set of coding standards checkers to check, specified as a `.xml` file. You can, in the same file, define a custom set of checkers for each of the coding standards that Polyspace supports. To create a file that defines a custom selection of coding standard checkers, in the Polyspace interface, select a coding standard on the **Coding Standards & Code Metrics** node of the **Configuration** pane and click **Edit**.

For more information, see `Set checkers by file (-checkers-selection-file)`.

Example: `opts.CodingRulesCodeMetrics.CheckersSelectionByFile = 'C:\ps_settings\coding_rules\custom_rules.xml'`

Data Types: `char`

`CodeMetrics` — Activate code metric calculations
false (default) | true

Activate code metric calculations, specified as true or false. If this property is turned off, Polyspace does not calculate code metrics even if you upload your results to Polyspace Metrics.

For more information about the code metrics, see `Calculate code metrics (-code-metrics)`.

If you assign a coding rules options object to this property, an XML file gets created automatically with the rules specified.

Example: `opts.CodingRulesCodeMetrics.CodeMetrics = true`

`EnableAcAgc` — Check MISRA AC AGC rules
false (default) | true

Check MISRA AC AGC rules, specified as true or false. To customize which rules are checked, use `AcAgcSubset`.

For more information about the MISRA AC AGC checker, see `Check MISRA AC AGC (-misra-ac-agc)`.

Example: `opts.CodingRulesCodeMetrics.EnableAcAgc = true;`

`EnableAutosarCpp14` — Check AUTOSAR C++ 14 rules
false (default) | true

This property affects Bug Finder only.

Check AUTOSAR C++ 14 rules, specified as true or false. To customize which rules are checked, use `AutosarCpp14`.

For more information about the AUTOSAR C++ 14 checker, see `Check AUTOSAR C++ 14 security checks (-autosar-cpp14)`.

Example: `opts.CodingRulesCodeMetrics.EnableAutosarCpp14 = true;`

`EnableCertC` — check CERT C rules and recommendations
false (default) | true

This property affects Bug Finder only.

Check CERT C rules and recommendations, specified as true or false. To customize which rules are checked, use `CertC`.

For more information about the CERT C checker, see `Check CERT-C security checks (-cert-c)`.

Example: `opts.CodingRulesCodeMetrics.EnableCertC = true;`

`EnableCertCpp` — check CERT C++ rules
false (default) | true

This property affects Bug Finder only.

Check CERT C++ rules, specified as true or false. To customize which rules are checked, use `CertCpp`.

For more information about the CERT C++ checker, see `Check CERT-C++ security checks (-cert-cpp)`.

Example: `opts.CodingRulesCodeMetrics.EnableCertCpp = true;`

`EnableCheckersSelectionByFile` — Check custom set of coding standard checkers
false (default) | true

Check custom set of coding standard checkers, specified as true or false. Use with `CheckersSelectionByFile` and these coding standards:

- `opts.CodingRulesCodeMetrics.AutosarCpp14='from-file'`
- `opts.CodingRulesCodeMetrics.CertC='from-file'`
- `opts.CodingRulesCodeMetrics.CertCpp='from-file'`
- `opts.CodingRulesCodeMetrics.Iso17961='from-file'`
- `opts.CodingRulesCodeMetrics.JsfSubset='from-file'`

- `opts.CodingRulesCodeMetrics.MisraC3Subset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraCSubset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraCppSubset='from-file'`

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCheckersSelectionByFile = true;`

`EnableCustomRules` — Check custom coding rules
false (default) | true

Check custom coding rules, specified as true or false. The file you specify with `CheckersSelectionByFile` defines the custom coding rules.

Use with `EnableCheckersSelectionByFile`.

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCustomRules = true;`

`EnableIso17961` — check ISO-17961 rules
false (default) | true

This property affects Bug Finder only.

Check ISO/IEC TS 17961 rules, specified as true or false. To customize which rules are checked, use `Iso17961`.

For more information about the ISO-17961 checker, see `Check ISO-17961 security checks (-iso-17961)`.

Example: `opts.CodingRulesCodeMetrics.EnableIso17961 = true;`

`EnableJsf` — Check JSF C++ rules
false (default) | true

Check JSF C++ rules, specified as true or false. To customize which rules are checked, use `JsfSubset`.

For more information, see `Check JSF C++ rules (-jsf-coding-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableJsf = true;`

`EnableMisraC` — Check MISRA C:2004 rules
false (default) | true

Check MISRA C:2004 rules, specified as true or false. To customize which rules are checked, use `MisraCSubset`.

For more information, see `Check MISRA C:2004 (-misra2)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraC = true;`

`EnableMisraC3` — Check MISRA C:2012 rules
false (default) | true

Check MISRA C:2012 rules, specified as true or false. To customize which rules are checked, use `MisraC3Subset`.

For more information about the MISRA C:2012 checker, see `Check MISRA C:2012 (-misra3)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraC3 = true;`

`EnableMisraCpp` — Check MISRA C++:2008 rules
false (default) | true

Check MISRA C++:2008 rules, specified as true or false. To customize which rules are checked, use `MisraCppSubset`.

For more information about the MISRA C++:2008 checker, see `Check MISRA C++ rules (-misra-cpp)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraCpp = true;`

`Iso17961` — Set of ISO-17961 rules to check
`'all'` (default) | `'decidable'` | `polyspace.CodingRulesOptions` object | `'from-file'`

This property affects Bug Finder only.

Set of ISO/IEC TS 17961 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check ISO-17961 security checks (-iso-17961)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check ISO/IEC TS 17961 rules, also set `EnableIso17961` to true.

Example: `opts.CodingRulesCodeMetrics.Iso17961 = 'all'`

Data Types: `char`

`JsfSubset` — Subset of JSF C++ rules to check
`'shall-rules'` (default) | `'shall-will-rules'` | `'all-rules'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of JSF C++ rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check JSF C++ rules (-jsf-coding-rules)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.

- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check JSF C++ rules, set `EnableJsf` to true.

Example: `opts.CodingRulesCodeMetrics.JsfSubset = 'all-rules'`

Data Types: `char`

`Misra3AgcMode` — Use the MISRA C:2012 categories for automatically generated code
false (default) | true

Use the MISRA C:2012 categories for automatically generated code, specified as true or false.

For more information, see `Use generated code requirements (-misra3-agc-mode)`.

Example: `opts.CodingRulesCodeMetrics.Misra3AgcMode = true;`

`MisraC3Subset` — Subset of MISRA C:2012 rules to check
`'mandatory-required'` (default) | `'mandatory'` | `'single-unit-rules'` | `'system-decidable-rules'` | `'all'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C:2012 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2012 (-misra3)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C:2012 rules, also set `EnableMisraC3` to true.

Example: `opts.CodingRulesCodeMetrics.MisraC3Subset = 'all'`

Data Types: `char`

`MisraCSubset` — Subset of MISRA C:2004 rules to check
`'required-rules'` (default) | `'single-unit-rules'` | `'system-decidable-rules'` | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C:2004 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2004 (-misra2)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C:2004 rules, also set `EnableMisraC` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCSubset = 'all-rules'`

Data Types: `char`

`MisraCppSubset` — Subset of MISRA C++ rules
`'required-rules'` (default) | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C++:2008 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C++ rules (-misra-cpp)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C++ rules, set `EnableMisraCpp` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCppSubset = 'all-rules'`

Data Types: `char`

EnvironmentSettings

`Dos` — Consider that file paths are in MS-DOS style
true (default) | false

Consider that file paths are in MS-DOS style, specified as true or false.

For more information, see `Code from DOS or Windows file system (-dos)`.

Example: `opts.EnvironmentSettings.Dos = true;`

`IncludeFolders` — Include folders needed for compilation
cell array of include folder paths

Include folders needed for compilation, specified as a cell array of the include folder paths.

To specify all subfolders of a folder, use folder path followed by **, for instance, `'C:\includes\**'`. The notation follows the syntax of the `dir` function. See also “Specify Multiple Source Files”.

For more information, see `-I`.

Example: `opts.EnvironmentSettings.IncludeFolders = {'/includes','/com1/inc'};`

Example: `opts.EnvironmentSettings.IncludeFolders = {'C:\project1\common\includes'};`

Data Types: `cell`

`Includes` — Files to be `#include`-ed by each C file
cell array of files

Files to be `#include`-ed by each C source file in the analysis, specified by a cell array of files.

For more information, see `Include (-include)`.

Example: `opts.EnvironmentSettings.Includes = {'/inc/inc_file.h','/inc/inc_math.h'}`

`NoExternC` — Ignore linking errors inside extern blocks
false (default) | true

Ignore linking errors inside extern blocks, specified as true or false.

For more information, see `Ignore link errors (-no-extern-c)`.

Example: `opts.EnvironmentSettings.NoExternC = false;`

`PostPreProcessingCommand` — Command or script to run on source files after preprocessing
character vector

Command or script to run on source files after preprocessing, specified as a character vector of the command to run.

For more information, see `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

Example: Linux — `opts.EnvironmentSettings.PostPreProcessingCommand = [pwd,'/replace_keyword.pl']`

Example: Windows — `opts.EnvironmentSettings.PostPreProcessingCommand = '"C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\replace_keyword.pl"'`

`StopWithCompileError` — Stop analysis if a file does not compile
false (default) | true

Stop analysis if a file does not compile, specified as true or false.

For more information, see `Stop analysis if a file does not compile (-stop-if-compile-error)`.

Example: `opts.EnvironmentSettings.StopWithCompileError = true;`

InputsStubbing

`DataRangeSpecifications` — Constrain global variables, function inputs, and return values of stubbed functions
file path

Constrain global variables, function inputs, and return values of stubbed functions specified by the path to an XML constraint file. For more information about the constraint file, see “Specify External Constraints”.

For more information about this option, see `Constraint setup (-data-range-specifications)`.

Example: `opts.InputsStubbing.DataRangeSpecifications = 'C:\project\constraint_file.xml'`

`DoNotGenerateResultsFor` — Files on which you do not want analysis results
`'include-folders'` (default) | `'all-headers'` | `'custom=`*`folder1[,folder2,...]`*`'`

Files on which you do not want analysis results, specified by `'include-folders'`, `'all-headers'`, or a character array beginning with `custom=` followed by a comma-separated list of file or folder names.

Use this option with `InputsStubbing.GenerateResultsFor`. For more information, see `Do not generate results for (-do-not-generate-results-for)`.

Example: `opts.InputsStubbing.DoNotGenerateResultsFor = 'custom=C:\project\file1.c,C:\project\file2.c'`

`GenerateResultsFor` — Files on which you want analysis results
`'source-headers'` (default) | `'all-headers'` | `'custom=`*`folder1[,folder2,...]`*`'`

Files on which you want analysis results, specified by `'source-headers'`, `'all-headers'`, or a character array beginning with `custom=` followed by a comma-separated list of file or folder names.

Use this option with `InputsStubbing.DoNotGenerateResultsFor`. For more information, see `Generate results for sources and (-generate-results-for)`.

Example: `opts.InputsStubbing.GenerateResultsFor = 'custom=C:\project\includes_common_1,C:\project\includes_common_2'`

`FunctionsToStub` — Functions to stub during analysis
cell array of function names

This property affects Code Prover analysis only.

Functions to stub during analysis, specified as a cell array of function names.

For more information, see `Functions to stub (-functions-to-stub)`.

Example: `opts.InputsStubbing.FunctionsToStub = {'func1', 'func2'}`

`NoDefInitGlob` — Consider global variables as uninitialized
false (default) | true

This property affects Code Prover analysis only.

Consider global variables as uninitialized, specified as true or false.

For more information, see `Ignore default initialization of global variables (-no-def-init-glob)`.

Example: `opts.InputsStubbing.NoDefInitGlob = true`

`NoStlStubs` — Do not use Polyspace implementations of functions in the Standard Template Library
false (default) | true

This property applies only to a Code Prover analysis of C++ code.

Do not use Polyspace implementations of functions in the Standard Template Library, specified as true or false.

For more information, see `No STL stubs (-no-stl-stubs)`.

Example: `opts.InputsStubbing.NoStlStubs = true`

`StubECoderLookupTables` — Specify that the analysis must stub functions in the generated code that use lookup tables
true (default) | false

This property applies only to a Code Prover analysis of code generated from models.

Specify that the analysis must stub functions in the generated code that use lookup tables. By replacing the functions with stubs, the analysis assumes more precise return values for the functions.

For more information, see `Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions)`.

Example: `opts.InputsStubbing.StubECoderLookupTables = true`

Macros

`DefinedMacros` — Macros to be replaced
cell array of macros

In preprocessed code, macros are replaced by the definition, specified in a cell array of macros and definitions. Specify the macro as `Macro=Value`. If you want Polyspace to ignore the macro, leave the `Value` blank. A macro with no equal sign replaces all instances of that macro by 1.

For more information, see `Preprocessor definitions (-D)`.

Example: `opts.Macros.DefinedMacros = {'uint32=int','name3=','var'}`

`UndefinedMacros` — Macros to undefine
cell array of macros

In preprocessed code, macros are undefined, specified by a cell array of macros to undefine.

For more information, see `Disabled preprocessor definitions (-U)`.

Example: `opts.Macros.DefinedMacros = {'name1','name2'}`

MergedComputingSettings

`AddToResultsRepositoryBugFinder` — Upload Bug Finder results to Polyspace Metrics web dashboard
false (default) | true

This property affects Bug Finder analysis only.

Upload Bug Finder analysis results to Polyspace Metrics web dashboard, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Upload results to Polyspace Metrics (-add-to-results-repository)`.

Example: `opts.MergedComputingSettings.AddToResultsRepositoryBugFinder = true;`

`AddToResultsRepositoryCodeProver` — Upload Code Prover results to Polyspace Metrics web dashboard
false (default) | true

This property affects Code Prover analysis only.

Upload Code Prover analysis results to Polyspace Metrics web dashboard, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Upload results to Polyspace Metrics (-add-to-results-repository)`.

Example: `opts.MergedComputingSettings.AddToResultsRepositoryCodeProver = true;`

`BatchBugFinder` — Send Bug Finder analysis to remote server
false (default) | true

This property affects Bug Finder analysis only.

Send Bug Finder analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`.

Example: `opts.MergedComputingSettings.BatchBugFinder = true;`

`BatchCodeProver` — Send Code Prover analysis to remote server
false (default) | true

This property affects Code Prover analysis only.

Send Code Prover analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`.

Example: `opts.MergedComputingSettings.BatchCodeProver = true;`

`FastAnalysis` — Run Bug Finder analysis using faster local mode
false (default) | true

This property affects Bug Finder analysis only.

Use fast analysis mode for Bug Finder analysis, specified as true or false.

For more information, see `Use fast analysis mode for Bug Finder (-fast-analysis)`.

Example: `opts.MergedComputingSettings.FastAnalysis = true;`

MergedReporting

`EnableReportGeneration` — Generate a report after the analysis
false (default) | true

After the analysis, generate a report, specified as true or false.

For more information, see `Generate report`.

Example: `opts.MergedReporting.EnableReportGeneration = true`

`ReportOutputFormat` — Output format of generated report
`'Word'` (default) | `'HTML'` | `'PDF'`

Output format of generated report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Output format (-report-output-format)`.

Example: `opts.MergedReporting.ReportOutputFormat = 'PDF'`

`BugFinderReportTemplate` — Template for generating Bug Finder analysis report
`'BugFinderSummary'` (default) | `'BugFinder'` | `'SecurityCWE'` | `'CodeMetrics'` | `'CodingStandards'`

This property affects a Bug Finder analysis only.

Template for generating analysis report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Bug Finder and Code Prover report (-report-template)`.

Example: `opts.MergedReporting.BugFinderReportTemplate = 'CodeMetrics'`

`CodeProverReportTemplate` — Template for generating Code Prover analysis report
`'Developer'` (default) | `'CallHierarchy'` | `'CodeMetrics'` | `'CodingStandards'` | `'DeveloperReview'` | `'Developer_withGreenChecks'` | `'Quality'` | `'VariableAccess'`

This property affects a Code Prover analysis only.

Template for generating analysis report, specified as one of the predefined report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Bug Finder and Code Prover report (-report-template)`.

Example: `opts.MergedReporting.CodeProverReportTemplate = 'CodeMetrics'`

Multitasking

`ArxmlMultitasking` — Specify path of ARXML files to parse for multitasking configuration
cell array of file paths

Specify the path to the ARXML files the software parses to set up your multitasking configuration.

To activate this option, specify `Multitasking.EnableExternalMultitasking` and set `Multitasking.ExternalMultitaskingType` to `autosar`.

For more information, see `ARXML files selection (-autosar-multitasking)`

Example: `opts.Multitasking.ArxmlMultitasking={'C:\Polyspace_Workspace\AUTOSAR\myFile.arxml'}`

`CriticalSectionBegin` — Functions that begin critical sections
cell array of critical section function names

Functions that begin critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionEnd`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

Example: `opts.Multitasking.CriticalSectionBegin = {'function1:cs1','function2:cs2'}`

`CriticalSectionEnd` — Functions that end critical sections
cell array of critical section function names

Functions that end critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionBegin`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

Example: `opts.Multitasking.CriticalSectionEnd = {'function1:cs1','function2:cs2'}`

`CyclicTasks` — Specify functions that represent cyclic tasks
cell array of function names

Specify functions that represent cyclic tasks.

To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Cyclic tasks (-cyclic-tasks)`.

Example: `opts.Multitasking.CyclicTasks = {'function1','function2'}`

`EnableConcurrencyDetection` — Enable automatic detection of certain families of threading functions
false (default) | true

This property affects Code Prover analysis only.

Enable automatic detection of certain families of threading functions, specified as true or false.

For more information, see `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

Example: `opts.Multitasking.EnableConcurrencyDetection = true`

`EnableExternalMultitasking` — Enable automatic multitasking configuration from external file definitions
false (default) | true

Enable multitasking configuration of your projects from external files you provide. Configure multitasking from ARXML files for an AUTOSAR project, or from OIL files for an OSEK project.

Activate this option to enable `Multitasking.ArxmlMultitasking` or `Multitasking.OsekMultitasking`.

For more information, see `OIL files selection (-osek-multitasking)` and `ARXML files selection (-autosar-multitasking)`.

Example: `opts.Multitasking.EnableExternalMultitasking = 1`

`EnableMultitasking` — Configure multitasking manually
false (default) | true

Configure multitasking manually by specifying `true`. This property activates the other manual, multitasking properties.

For more information, see `Configure multitasking manually`.

Example: `opts.Multitasking.EnableMultitasking = 1`

`EntryPoints` — Functions that serve as entry-points to your multitasking application
cell array of entry-point function names

Functions that serve as entry-points to your multitasking application specified as a cell array of entry-point function names. To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Tasks (-entry-points)`.

Example: `opts.Multitasking.EntryPoints = {'function1','function2'}`

`ExternalMultitaskingType` — Specify type of file to parse for multitasking configuration
`'osek'` (default) | `'autosar'`

Specify the type of file the software parses to set up your multitasking configuration:

- For `osek` type, the analysis looks for OIL files in the file or folder paths that you specify.
- For `autosar` type, the analysis looks for ARXML files in the file paths that you specify.

To activate this option, specify `Multitasking.EnableExternalMultitasking`.

For more information, see `OIL files selection (-osek-multitasking)` and `ARXML files selection (-autosar-multitasking)`.

Example: `opts.Multitasking.ExternalMultitaskingType = 'autosar'`

`Interrupts` — Specify functions that represent nonpreemptable interrupts
cell array of function names

Specify functions that represent nonpreemptable interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Interrupts (-interrupts)`.

Example: `opts.Multitasking.Interrupts = {'function1','function2'}`

`InterruptsDisableAll` — Specify routine that disable interrupts
cell array with one function name

This property affects Bug Finder analysis only.

Specify function that disables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Example: `opts.Multitasking.InterruptsDisableAll = {'function'}`

`InterruptsEnableAll` — Specify routine that reenable interrupts
cell array with one function name

This property affects Bug Finder analysis only.

Specify function that reenables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Example: `opts.Multitasking.InterruptsEnableAll = {'function'}`

`OsekMultitasking` — Specify path of OIL files to parse for multitasking configuration
`'auto'` (default) | `'custom=`*`folder1[,folder2,...]`*`'`

Specify the path to the OIL files the software parses to set up your multitasking configuration:

- In the mode specified with `'auto'`, the analysis uses OIL files in your project source and include folders, but not their subfolders.
- In the mode specified with `'custom=`*`folder1[,folder2,...]`*`'`, the analysis uses the OIL files at the specified path, and the path subfolders.

To activate this option, specify `Multitasking.EnableExternalMultitasking` and set `Multitasking.ExternalMultitaskingType` to `osek`.

For more information, see `OIL files selection (-osek-multitasking)`

Example: `opts.Multitasking.OsekMultitasking = 'custom=file_path, dir_path'`

`TemporalExclusion` — Entry-point functions that cannot execute concurrently
cell array of entry-point function names

Entry-point functions that cannot execute concurrently specified as a cell array of entry-point function names. Each set of exclusive tasks is one cell array entry with functions separated by spaces. To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Temporally exclusive tasks (-temporal-exclusions-file)`.

Example: `opts.Multitasking.TemporalExclusion = {'function1 function2', 'function3 function4 function5'}` where function1 and function2 are temporally exclusive, and function3, function4, and function 5 are temporally exclusive.

Precision (Affects Code Prover Only)

`ContextSensitivity` — Store call context information to identify function call that caused errors

`'none'` (default) | `'auto'` | `'custom=`*`function1[,function2,...]`*`'`

This property affects Code Prover analysis only.

Store call context information to identify a function call that caused errors, specified as `none`, `auto`, or as a character array beginning with `custom=` followed by a list of comma-separated function names.

For more information, see `Sensitivity context (-context-sensitivity)`.

Example: `opts.Precision.ContextSensitivity = 'auto'`

Example: `opts.Precision.ContextSensitivity = 'custom=func1'`

`ModulesPrecision` — Source files you want to verify at higher precision

cell array of file names and precision levels

This property affects Code Prover analysis only.

Source files that you want to verify at higher precision, specified as a cell array of file names without the extension and precision levels using this syntax: *`filename`*`:O`*`level`*

For more information, see `Specific precision (-modules-precision)`.

Example: `opts.Precision.ModulesPrecision = {'file1:O0', 'file2:O3'}`

`OLevel` — Precision level for the verification

2 (default) | 0 | 1 | 3

This property affects Code Prover analysis only.

Precision level for the verification, specified as 0, 1, 2, or 3.

For more information, see `Precision level (-O)`.

Example: `opts.Precision.OLevel = 3`

`PathSensitivityDelta` — Avoid certain verification approximations for code with fewer lines

positive integer

This property affects Code Prover analysis only.

Avoid certain verification approximations for code with fewer lines, specified as a positive integer representing how sensitive the analysis is. Higher values can increase verification time exponentially.

For more information, see `Improve precision of interprocedural analysis (-path-sensitivity-delta)`.

Example: `opts.Precision.PathSensitivityDelta = 2`

`Timeout` — Time limit on your verification
character vector

This property affects Code Prover analysis only.

Time limit on your verification, specified as a character vector of time in hours.

For more information, see `Verification time limit (-timeout)`.

Example: `opts.Precision.Timeout = '5.75'`

`To` — Number of times the verification process runs
`'Software Safety Analysis level 2'` (default) | `'Software Safety Analysis level 0'` | `'Software Safety Analysis level 1'` | `'Software Safety Analysis level 3'` | `'Software Safety Analysis level 4'` | `'Source Compliance Checking'` | `'other'`

This property affects Code Prover analysis only.

Number of times the verification process runs, specified as one of the preset analysis levels.

For more information, see `Verification level (-to)`.

Example: `opts.Precision.To = 'Software Safety Analysis level 3'`

Scaling (Affects Code Prover Only)

`Inline` — Functions on which separate results must be generated for each function call
cell array of function names

This property affects Code Prover analysis only.

Functions on which separate results must be generated for each function call, specified as a cell array of function names.

For more information, see `Inline (-inline)`.

Example: `opts.Scaling.Inline = {'func1','func2'}`

`KLimiting` — Limit depth of analysis for nested structures
positive integer

This property affects Code Prover analysis only.

Limit depth of analysis for nested structures, specified as a positive integer indicating how many levels into a nested structure to verify.

For more information, see `Depth of verification inside structures (-k-limiting)`.

Example: `opts.Scaling.KLimiting = 3`

TargetCompiler

`Compiler` — Compiler that builds your source code
`'generic'` (default) | `'gnu3.4'` | `'gnu4.6'` | `'gnu4.7'` | `'gnu4.8'` | `'gnu4.9'` | `'gnu5.x'` | `'gnu6.x'` | `'gnu7.x'` | `'clang3.x'` | `'clang4.x'` | `'clang5.x'` | `'visual9.0'` | `'visual10'` | `'visual11.0'` | `'visual12.0'` | `'visual14.0'` | `'visual15.x'` | `'keil'` | `'iar'` | `'armcc'` | `'armclang'` | `'codewarrior'` | `'diab'` | `'greenhills'` | `'iar-ew'` | `'renesas'` | `'tasking'` | `'ti'`

Compiler that builds your source code.

For more information, see `Compiler (-compiler)`.

Example: `opts.TargetCompiler.Compiler = 'Visual11.0'`

`CppVersion` — Specify C++11 standard version followed in code
`'defined-by-compiler'` (default) | `'cpp03'` | `'cpp11'` | `'cpp14'`

Specify C++ standard version followed in code, specified as a character vector.

For more information, see `C++ standard version (-cpp-version)`.

Example: `opts.TargetCompiler.CppVersion = 'cpp11';`

`CVersion` — Specify C standard version followed in code
`'defined-by-compiler'` (default) | `'c90'` | `'c99'` | `'c11'`

Specify C standard version followed in code, specified as a character vector.

For more information, see `C standard version (-c-version)`.

Example: `opts.TargetCompiler.CVersion = 'c90';`

`DivRoundDown` — Round down quotients from division or modulus of negative numbers
false (default) | true

Round down quotients from division or modulus of negative numbers, specified as true or false.

For more information, see `Division round down (-div-round-down)`.

Example: `opts.TargetCompiler.DivRoundDown = true`

`EnumTypeDefinition` — Base type representation of enum
`'defined-by-compiler'` (default) | `'auto-signed-first'` | `'auto-unsigned-first'`

Base type representation of enum, specified by an allowed base-type set. For more information about the different values, see `Enum type definition (-enum-type-definition)`.

Example: `opts.TargetCompiler.EnumTypeDefinition = 'auto-unsigned-first'`

`IgnorePragmaPack` — Ignore #pragma pack directives
false (default) | true

Ignore #pragma pack directives, specified as true or false.

For more information, see `Ignore pragma pack directives (-ignore-pragma-pack)`.

Example: `opts.TargetCompiler.IgnorePragmaPack = true`

Language — Language of analysis
'C-CPP' (default) | 'C' | 'CPP'

This property is read-only.

Language of the analysis, specified during the object construction. This value changes which properties appear.

For more information, see Source code language (-lang).

LogicalSignedRightShift — Treatment of signed bit on signed variables
'Arithmetical' (default) | 'Logical'

Treatment of signed bit on signed variables, specified as Arithmetical or Logical. For more information, see Signed right shift (-logical-signed-right-shift).

Example: opts.TargetCompiler.LogicalSignedRightShift = 'Logical'

NoUliterals — Do not use predefined typedefs for char16_t or char32_t
false (default) | true

Do not use predefined typedefs for char16_t or char32_t, specified as true or false. For more information, see Block char16/32_t types (-no-uliterals).

Example: opts.TargetCompiler.NoUliterals = true

PackAlignmentValue — Default structure packing alignment
'defined-by-compiler' (default) | '1' | '2' | '4' | '8' | '16'

Default structure packing alignment, specified as 'defined-by-compiler', '1', '2', '4', '8', or '16'. This property is available only for Visual C++ code.

For more information, see Pack alignment value (-pack-alignment-value).

Example: opts.TargetCompiler.PackAlignmentValue = '4'

SfrTypes — sfr types
cell array of sfr keywords

sfr types, specified as a cell array of sfr keywords using the syntax *sfr_name=size_in_bits*. For more information, see Sfr type support (-sfr-types).

This option only applies when you set TargetCompiler.Compiler to keil or iar.

Example: opts.TargetCompiler.SfrTypes = {'sfr32=32'}

SizeTTypeIs — Underlying type of size_t
'defined-by-compiler' (default) | 'unsigned-int' | 'unsigned-long' | 'unsigned-long-long'

Underlying type of size_t, specified as 'defined-by-compiler', 'unsigned-int', 'unsigned-long', or 'unsigned-long-long'. See Management of size_t (-size-t-type-is).

Example: opts.TargetCompiler.SizeTTypeIs = 'unsigned-long'

Target — Target processor
'i386' (default) | 'arm' | 'arm64' | 'avr' | 'c-167' | 'c166' | 'c18' | 'c28x' | 'c6000' | 'coldfire' | 'hc08' | 'hc12' | 'm68k' | 'mcore' | 'mips' | 'mpc5xx' | 'msp430' | 'necv850'

| 'powerpc' | 'powerpc64' | 'rh850' | 'rl78' | 'rx' | 's12z' | 'sharc21x61' | 'sparc' | 'superh' | 'tms320c3x' | 'tricore' | 'x86_64' | generic target object

Set size of data types and endianness of processor, specified as one of the predefined target processors or a generic target object.

For more information about the predefined processors, see `Target processor type (-target)`.

For more information about creating a generic target, see `polyspace.GenericTargetOptions`.

Example: `opts.TargetCompiler.Target = 'hc12'`

`WcharTTypeIs` — Underlying type of `wchar_t`
`'defined-by-compiler'` (default) | `'signed-short'` | `'unsigned-short'` | `'signed-int'` | `'unsigned-int'` | `'signed-long'` | `'unsigned-long'`

Underlying type of `wchar_t`, specified as `'defined-by-compiler'`, `'signed-short'`, `'unsigned-short'`, `'signed-int'`, `'unsigned-int'`, `'signed-long'`, or `'unsigned-long'`. See `Management of wchar_t (-wchar-t-type-is)`.

Example: `opts.TargetCompiler.WcharTTypeIs = 'unsigned-int'`

VerificationAssumption (Affects Code Prover Only)

`ConsiderVolatileQualifierOnFields` — Assume that volatile qualified structure fields can have all possible values at any point in code
false (default) | true

This property affects Code Prover analysis only.

Assume that volatile qualified structure fields can have all possible values at any point in code.

For more information, see `Consider volatile qualifier on fields (-consider-volatile-qualifier-on-fields)`.

Example: `opts.VerificationAssumption.ConsiderVolatileQualifierOnFields = true`

`ConstraintPointersMayBeNull` — Specify that environment pointers can be NULL unless constrained otherwise
false (default) | true

This property affects Code Prover analysis only.

Specify that environment pointers can be NULL unless constrained otherwise.

For more information, see `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`.

Example: `opts.VerificationAssumption.ConstraintPointersMayBeNull = true`

`FloatRoundingMode` — Rounding modes to consider when determining the results of floating-point arithmetic
`to-nearest` (default) | `all`

This property affects Code Prover analysis only.

Rounding modes to consider when determining the results of floating-point arithmetic, specified as `to-nearest` or `all`.

For more information, see `Float rounding mode (-float-rounding-mode)`.

Example: `opts.VerificationAssumption.FloatRoundingMode = 'all'`

`RespectTypesInFields` — Do not cast nonpointer fields of a structure to pointers
false (default) | true

This property affects Code Prover analysis only.

Do not cast nonpointer fields of a structure to pointers, specified as true or false.

For more information, see `Respect types in fields (-respect-types-in-fields)`.

Example: `opts.VerificationAssumption.RespectTypesInFields = true`

`RespectTypesInGlobals` — Do not cast nonpointer global variables to pointers
false (default) | true

This property affects Code Prover analysis only.

Do not cast nonpointer global variables to pointers, specified as true or false.

For more information, see `Respect types in global variables (-respect-types-in-globals)`.

Example: `opts.VerificationAssumption.RespectTypesInGlobals = true`

Other Properties

`Author` — Project author
username of current user (default) | character vector

Name of project author, specified as a character vector.

For more information, see `-author`.

Example: `opts.Author = 'JaneDoe'`

`ImportComments` — Import comments and justifications from previous analysis
character vector

To import comments and justifications from a previous analysis, specify the path to the results folder of the previous analysis.

You can also point to a previous results folder to see only new results compared to the previous run. See “Compare Results from Different Polyspace Runs by Using MATLAB Scripts”.

For more information, see `-import-comments`

Example: `opts.ImportComments = fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example','Module_1','BF_Result')`

`Prog` — Project name
`PolyspaceProject` (default) | character vector

Project name, specified as a character vector.

For more information, see `-prog`.

Example: `opts.Prog = 'myProject'`

`ResultsDir` — Location to store results
folder path

Location to store results, specified as a folder path. By default, the results are stored in the current folder.

For more information, see `-results-dir`.

You can also create a separate results folder for each new run. See “Compare Results from Different Polyspace Runs by Using MATLAB Scripts”.

Example: `opts.ResultsDir = 'C:\project\myproject\results\'`

`Sources` — Source files
cell array of files

Source files to analyze, specified as a cell array of files.

To specify all files in a folder, use folder path followed by *, for instance, `'C:\src\*'`. To specify all files in a folder and its subfolders, use folder path followed by **, for instance, `'C:\src\**'`. The notation follows the syntax of the `dir` function. See also “Specify Multiple Source Files”.

For more information, see `-sources`.

Example: `opts.Sources = {'file1.c', 'file2.c', 'file3.c'}`

Example: `opts.Sources = {'project/src1/file1.c', 'project/src2/file2.c', 'project/src3/file3.c'}`

`Version` — Project version number
`'1.0'` (default) | character array of a number

Version number of project, specified as a character array of a number. This option is useful if you upload your results to Polyspace Metrics. If you increment version numbers each time that you reanalyze your object, you can compare the results from two versions in Polyspace Metrics.

For more information, see `-v[ersion]`.

Example: `opts.Version = '2.3'`

See Also

Topics
“Analysis Options”

Introduced in R2017a

polyspace.ModelLinkOptions Properties

Customize Polyspace analysis of generated code with options object properties

Description

To customize your Polyspace analysis of generated code, modify the `polyspace.ModelLinkOptions` object properties. Each property corresponds to an analysis option on the **Configuration** pane in the Polyspace user interface.

The properties are grouped using the same categories as the **Configuration** pane. This page only shows what values each property can take. For details about:

- The different options, see the analysis options reference pages.
- How to create and use the object, see `polyspace.ModelLinkOptions`.

  The same properties are also available with the deprecated classes `polyspace.ModelLinkBugFinderOptions` and `polyspace.ModelLinkCodeProverOptions`.

Each property description below also highlights if the option affects only one of Bug Finder or Code Prover.

Note Some options might not be available depending on the language setting of the object. You can set the source code language (`Language`) to `'C'`, `'CPP'` or `'C-CPP'` during object creation, but cannot change it later.

Properties

Advanced

`Additional` — Additional flags for analysis

character vector

Additional flags for analysis specified as a character vector.

For more information, see `Other`.

Example: `opts.Advanced.Additional = '-extra-flags -option -extra-flags value'`

`PostAnalysisCommand` — Command or script software should execute after analysis finishes

character vector

Command or script software should execute after analysis finishes, specified as a character vector.

For more information, see `Command/script to apply after the end of the code verification (-post-analysis-command)`.

Example: `opts.Advanced.PostAnalysisCommand = '"C:\Program Files\perl\win32\bin\perl.exe" "C:\My_Scripts\send_email"'`

`AutomaticOrangeTester` — Run the Automatic Orange Tester
false (default) | true

This property affects Code Prover analysis only.

Run the Automatic Orange Tester after verification, specified as true or false.

For more information, see `Automatic Orange Tester (-automatic-orange-tester)`.

Example: `opts.Advanced.AutomaticOrangeTester = true`

`AutomaticOrangeTesterLoopMaxIteration` — Number of loop iterations after which Automatic Orange Tester considers infinite loop
1000 (default) | positive integer

This property affects Code Prover analysis only.

Number of loop iterations after which Automatic Orange Tester considers the test an infinite loop, specified as a positive integer, maximum of 1000.

For more information, see `Maximum loop iterations (-automatic-orange-tester-loop-max-iteration)`.

Example: `opts.Advanced.AutomaticOrangeTesterLoopMaxIteration = 500`

`AutomaticOrangeTesterTestsNumber` — Number of tests that Automatic Orange Tester must run
500 (default) | positive integer

This property affects Code Prover analysis only.

Number of tests that Automatic Orange Tester must run, specified as a positive integer, maximum of 100,000.

For more information, see `Number of automatic tests (-automatic-orange-tester-tests-number)`.

Example: `opts.Advanced.AutomaticOrangeTesterTestsNumber = 1000`

`AutomaticOrangeTesterTimeout` — Time in seconds allowed for a single test in Automatic Orange Tester
5 (default) | positive integer

This property affects Code Prover analysis only.

Time in seconds allowed for a single test in Automatic Orange Tester, specified as a positive integer, maximum of 60.

For more information, see `Maximum test time (-automatic-orange-tester-timeout)`.

Example: `opts.Advanced.AutomaticOrangeTesterTimeout = 10`

BugFinderAnalysis (Affects Bug Finder Only)

`CheckersList` — List of custom checkers to activate
`polyspace.DefectsOptions` object | cell array of defect acronyms

This property affects Bug Finder analysis only.

List of custom checkers to activate specified by using the name of a `polyspace.DefectsOptions` object or a cell array of defect acronyms. To use this custom list in your analysis, set `CheckersPreset` to `custom`.

For more information, see `polyspace.DefectsOptions`.

Example: `defects = polyspace.DefectsOptions;`
`opts.BugFinderAnalysis.CheckersList = defects`

Example: `opts.BugFinderAnalysis.CheckersList = {'INT_ZERO_DIV','FLOAT_ZERO_DIV'}`

`CheckersPreset` — Subset of Bug Finder defects
`'default'` (default) | `'all'` | `'CWE'` | `'custom'`

This property affects Bug Finder analysis only.

Preset checker list, specified as a character vector of one of the preset options: `'default'`, `'all'`, `'CWE'`,or `'custom'`. To use `'custom'`, specify a value for the property `BugFinderAnalysis.CheckersList`.

For more information, see `Find defects (-checkers)`.

Example: `opts.BugFinderAnalysis.CheckersPreset = 'all'`

`ChecksUsingSystemInputValues` — Activate stricter checks for system inputs
false (default) | true

This property affects Bug Finder analysis only.

Activate stricter checks that consider all possible value for:

- Global variables.
- Reads of volatile variables.
- Returns of stubbed functions.
- Inputs to functions specified with **SystemInputsFrom**.

The analysis considers all possible values for a subset of **Numerical** and **Static memory** defects.

This property is equivalent to the **Run stricter checks considering all values of system inputs** check box in the Polyspace interface.

For more information, see `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`

Example: `opts.BugFinderAnalysis.ChecksUsingSystemInputValues = true`

`EnableCheckers` — Activate defect checking
true (default) | false

This property affects Bug Finder analysis only.

Activate defect checking, specified as true or false. Setting this property to false disables all defects. If you want to disable defect checking but still get results, turn on coding rules checking or code metric checking.

This property is equivalent to the **Find defects** check box in the Polyspace interface.

Example: `opts.BugFinderAnalysis.EnableCheckers = false`

`SystemInputsFrom` — List of functions for which you run stricter checks
`'auto'` (default) | `'uncalled'` | `'all'` | `'custom'`

This property affects Bug Finder analysis only.

Functions for which you want to run stricter checks that consider all possible values of the function inputs. Specify the list of functions as `'auto'`, `'uncalled'`, `'all'`, or as a character array beginning with `custom=` followed by a comma-separated list of function names.

To enable this option, set `BugFinderAnalysis.ChecksUsingSystemInputValues = true`.

For more information, see `Consider inputs to these functions (-system-inputs-from)`

Example: `opts.BugFinderAnalysis.SystemInputsFrom = 'custom=foo,bar'`

ChecksAssumption (Affects Code Prover Only)

`AllowNegativeOperandInShift` — Allow left shift operations on a negative number
true (default) | false

This property affects Code Prover analysis only.

Allow left shift operations on a negative number, specified as true or false.

For more information, see `Allow negative operand for left shifts (-allow-negative-operand-in-shift)`.

Example: `opts.ChecksAssumption.AllowNegativeOperandInShift = true`

`AllowNonFiniteFloats` — Incorporate infinities and/or NaNs
false (default) | true

This property affects Code Prover analysis only.

Incorporate infinities and/or NaNs, specified as true or false.

For more information, see `Consider non finite floats (-allow-non-finite-floats)`.

Example: `opts.ChecksAssumption.AllowNonFiniteFloats = true`

`AllowPtrArithOnStruct` — Allow arithmetic on pointer to a structure field so that it points to another field
false (default) | true

This property affects Code Prover analysis only.

Allow arithmetic on pointer to a structure field so that it points to another field, specified as true or false.

For more information, see `Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)`.

Example: `opts.ChecksAssumption.AllowPtrArithOnStruct = true`

`CheckInfinite` — Detect floating-point operations that result in infinities
`'allow'` (default) | `'warn-first'` | `'forbid'`

This property affects Code Prover analysis only.

Detect floating-point operations that result in infinities.

To activate this option, specify `ChecksAssumption.AllowNonFiniteFloats`.

For more information, see `Infinities (-check-infinite)`.

Example: `opts.ChecksAssumption.CheckInfinite = 'forbid'`

`CheckNan` — Detect floating-point operations that result in NaN-s
`'allow'` (default) | `'warn-first'` | `'forbid'`

This property affects Code Prover analysis only.

Detect floating-point operations that result in NaN-s.

To activate this option, specify `ChecksAssumption.AllowNonFiniteFloats`.

For more information, see `NaNs (-check-nan)`.

Example: `opts.ChecksAssumption.CheckNan = 'forbid'`

`CheckSubnormal` — Detect operations that result in subnormal floating point values
`'allow'` (default) | `'warn-first'` | `'warn-all'` | `'forbid'`

This property affects Code Prover analysis only.

Detect operations that result in subnormal floating point values.

For more information, see `Subnormal detection mode (-check-subnormal)`.

Example: `opts.ChecksAssumption.CheckSubnormal = 'forbid'`

`DetectPointerEscape` — Find cases where a function returns a pointer to one of its local variables
false (default) | true

This property affects Code Prover analysis only.

Find cases where a function returns a pointer to one of its local variables, specified as true or false.

For more information, see `Detect stack pointer dereference outside scope (-detect-pointer-escape)`.

Example: `opts.ChecksAssumption.DetectPointerEscape = true`

`DisableInitializationChecks` — Disable checks for noninitialized variables and pointers
false (default) | true

This property affects Code Prover analysis only.

Disable checks for noninitialized variables and pointers, specified as true or false.

For more information, see `Disable checks for non-initialization (-disable-initialization-checks)`.

Example: `opts.ChecksAssumption.DisableInitializationChecks = true`

`PermissiveFunctionPointer` — Allow type mismatch between function pointers and the functions they point to
false (default) | true

This property affects Code Prover analysis only.

Allow type mismatch between function pointers and the functions they point to, specified as true or false.

For more information, see `Permissive function pointer calls (-permissive-function-pointer)`.

Example: `opts.ChecksAssumption.PermissiveFunctionPointer = true`

`SignedIntegerOverflows` — Behavior of signed integer overflows
`'warn-with-wrap-around'` (default) | `'forbid'` | `'allow'`

This property affects Code Prover analysis only.

Enable the check for signed integer overflows and the assumptions to make following an overflow specified as `'forbid'`, `'allow'`, or `'warn-with-wrap-around'`.

For more information, see `Overflow mode for signed integer (-signed-integer-overflows)`.

Example: `opts.ChecksAssumption.SignedIntegerOverflows = 'warn-with-wrap-around'`

`SizeInBytes` — Allow a pointer with insufficient memory buffer to point to a structure
false (default) | true

This property affects Code Prover analysis only.

Allow a pointer with insufficient memory buffer to point to a structure, specified as true or false.

For more information, see `Allow incomplete or partial allocation of structures (-size-in-bytes)`.

Example: `opts.ChecksAssumption.SizeInBytes = true`

`UncalledFunctionCheck` — Detect functions that are not called directly or indirectly from main or another entry-point function
`'none'` (default) | `'never-called'` | `'called-from-unreachable'` | `'all'`

This property affects Code Prover analysis only.

Detect functions that are not called directly or indirectly from main or another entry-point function, specified as `'none'`, `'never-called'`, `'called-from-unreachable'`, or `'all'`.

For more information, see `Detect uncalled functions (-uncalled-function-checks)`.

Example: `opts.ChecksAssumption.UncalledFunctionCheck = 'all'`

`UnsignedIntegerOverflows` — Behavior of unsigned integer overflows
`'allow'` (default) | `'forbid'` | `'warn-with-wrap-around'`

This property affects Code Prover analysis only.

Enable the check for unsigned integer overflows and the assumptions to make following an overflow, specified as `'forbid'`, `'allow'`, or `'warn-with-wrap-around'`.

For more information, see `Overflow mode for unsigned integer (-unsigned-integer-overflows)`.

Example: `opts.ChecksAssumption.UnsignedIntegerOverflows = 'allow'`

CodeProverVerification (Affects Code Prover only)

`ClassAnalyzer` — Classes that you want to verify
`'none'` (default) | `'all'` | `'custom=`*`class1[,class2,...]`*`'`

This property affects Code Prover analysis only.

Classes that you want to verify, specified as `'all'`, `'none'`, or as a character array beginning with `custom=` followed by a comma-separated list of class names.

For more information, see `Class (-class-analyzer)`.

Example: `opts.CodeProverVerification.ClassAnalyzer = 'none'`

`FunctionsCalledAfterLoop` — Functions that the generated main must call after the cyclic code loop
cell array of function names

This property affects Code Prover analysis only.

Functions that the generated main must call after the cyclic code loop, specified as a cell array of function names.

For more information, see `Termination functions (-functions-called-after-loop)`.

Example: `opts.CodeProverVerification.FunctionsCalledAfterLoop = {'func1','func2'}`

`FunctionsCalledBeforeLoop` — Functions that the generated main must call before the cyclic code loop
cell array of function names

This property affects Code Prover analysis only.

Model Link only. Functions that the generated main must call before the cyclic code loop, specified as a cell array of function names.

For more information, see `Initialization functions (-functions-called-before-loop))`.

Example: `opts.CodeProverVerification.FunctionsCalledBeforeLoop = {'func1','func2'}`

`FunctionsCalledInLoop` — Functions that the generated main must call in the cyclic code loop
`'none'` (default) | `'all'` | `'custom=`*`function1[,function2,...]`*`'`

This property affects Code Prover analysis only.

Functions that the generated main must call in the cyclic code loop, specified as `'none'`, `'all'`, or as a character array beginning with `custom=` followed by a comma-separated list of function names..

For more information, see `Step functions (-functions-called-in-loop)`.

Example: `opts.CodeProverVerification.FunctionsCalledInLoop = 'all'`

`MainGenerator` — Generate a main function if it is not present in source files
true (default) | false

This property affects Code Prover analysis only.

Generate a main function if it is not present in source files, specified as true or false.

For more information, see `Verify module or library (-main-generator)`.

Example: `opts.CodeProverVerification.MainGenerator = false`

`VariablesWrittenBeforeLoop` — Variables that the generated main must initialize before the cyclic code loop
`'none'` (default) | `'all'` | `'custom=`*`variable1[,variable2,...]`*`'`

This property affects Code Prover analysis only.

Variables that the generated main must initialize before the cyclic code loop, specified as `'none'`, `'all'`, or as a character array beginning with `custom=` followed by a comma-separated list of variable names.

For more information, see `Parameters (-variables-written-before-loop)`.

Example: `opts.CodeProverVerification.VariablesWrittenBeforeLoop = 'all'`

`VariablesWrittenInLoop` — Variables that the generated main must initialize in the cyclic code loop
`'none'` (default) | `'all'` | `'custom=`*`variable1[,variable2,...]`*`'`

This property affects Code Prover analysis only.

Variables that the generated main must initialize in the cyclic code loop, specified as `'none'`, `'all'`, or as a character array beginning with `custom=` followed by a comma-separated list of variable names.

For more information, see `Inputs (-variables-written-in-loop)`.

Example: `opts.CodeProverVerification.VariablesWrittenInLoop = 'all'`

CodingRulesCodeMetrics

`AcAgcSubset` — Subset of MISRA AC AGC rules to check
`'OBL-rules'` (default) | `'OBL-REC-rules'` | `'single-unit-rules'` | `'system-decidable-rules'` | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA AC AGC rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA AC AGC (-misra-ac-agc)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA AC AGC rules, also set `EnableAcAgc` to true.

Example: `opts.CodingRulesCodeMetrics.AcAgcSubset = 'all-rules'`

Data Types: `char`

`AllowedPragmas` — Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied
cell array of character vectors

Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied, specified as a cell array of character vectors. This property affects only MISRA C:2004 or MISRA AC AGC rule checking.

For more information, see `Allowed pragmas (-allowed-pragmas)`.

Example: `opts.CodingRulesCodeMetrics.AllowedPragmas = {'pragma_01','pragma_02'}`

Data Types: `cell`

`AutosarCpp14` — Set of AUTOSAR C++ 14 rules to check
`'all'` (default) | `'required'` | `'automated'` | `polyspace.CodingRulesOptions` object | `'from-file'`

This property affects Bug Finder only.

Set of AUTOSAR C++ 14 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check AUTOSAR C++ 14 security checks (-autosar-cpp14)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check AUTOSAR C++ 14 rules, also set `EnableAutosarCpp14` to true.

Example: `opts.CodingRulesCodeMetrics.AutosarCpp14 = 'all'`

Data Types: `char`

`BooleanTypes` — Data types the coding rule checker must treat as effectively Boolean
cell array of character vectors

Data types that the coding rule checker must treat as effectively Boolean, specified as a cell array of character vectors.

For more information, see `Effective boolean types (-boolean-types)`.

Example: `opts.CodingRulesCodeMetrics.BooleanTypes = {'boolean1_t','boolean2_t'}`

Data Types: `cell`

`CertC` — Set of CERT C rules and recommendations to check
`'all'` (default) | `'publish-2016'` | `'all-rules'` | `polyspace.CodingRulesOptions` object | `'from-file'`

This property affects Bug Finder only.

Set of CERT C rules and recommendations to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check CERT-C security checks (-cert-c)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `from-file` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check CERT C rules and recommendations, also set `EnableCertC` to true.

Example: `opts.CodingRulesCodeMetrics.CertC = 'all'`

Data Types: `char`

`CertCpp` — Set of CERT C++ rules to check
`'all'` (default) | `polyspace.CodingRulesOptions` object | `'from-file'`

This property affects Bug Finder only.

Set of CERT C++ rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check CERT-C++ security checks (-cert-cpp)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.

- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check CERT C++ rules, also set `EnableCertCpp` to true.

Example: `opts.CodingRulesCodeMetrics.CertCpp = 'all'`

Data Types: `char`

`CheckersSelectionByFile` — File that defines custom set of coding standard checkers
full file path of `.xml` file

File where you define a custom set of coding standards checkers to check, specified as a `.xml` file. You can, in the same file, define a custom set of checkers for each of the coding standards that Polyspace supports. To create a file that defines a custom selection of coding standard checkers, in the Polyspace interface, select a coding standard on the **Coding Standards & Code Metrics** node of the **Configuration** pane and click **Edit**.

For more information, see `Set checkers by file (-checkers-selection-file)`.

Example: `opts.CodingRulesCodeMetrics.CheckersSelectionByFile = 'C:\ps_settings\coding_rules\custom_rules.xml'`

Data Types: `char`

`CodeMetrics` — Activate code metric calculations
false (default) | true

Activate code metric calculations, specified as true or false. If this property is turned off, Polyspace does not calculate code metrics even if you upload your results to Polyspace Metrics.

For more information about the code metrics, see `Calculate code metrics (-code-metrics)`.

If you assign a coding rules options object to this property, an XML file gets created automatically with the rules specified.

Example: `opts.CodingRulesCodeMetrics.CodeMetrics = true`

`EnableAcAgc` — Check MISRA AC AGC rules
false (default) | true

Check MISRA AC AGC rules, specified as true or false. To customize which rules are checked, use `AcAgcSubset`.

For more information about the MISRA AC AGC checker, see `Check MISRA AC AGC (-misra-ac-agc)`.

Example: `opts.CodingRulesCodeMetrics.EnableAcAgc = true;`

`EnableAutosarCpp14` — Check AUTOSAR C++ 14 rules
false (default) | true

This property affects Bug Finder only.

Check AUTOSAR C++ 14 rules, specified as true or false. To customize which rules are checked, use `AutosarCpp14`.

For more information about the AUTOSAR C++ 14 checker, see `Check AUTOSAR C++ 14 security checks (-autosar-cpp14)`.

Example: `opts.CodingRulesCodeMetrics.EnableAutosarCpp14 = true;`

`EnableCertC` — check CERT C rules and recommendations
false (default) | true

This property affects Bug Finder only.

Check CERT C rules and recommendations, specified as true or false. To customize which rules are checked, use `CertC`.

For more information about the CERT C checker, see `Check CERT-C security checks (-cert-c)`.

Example: `opts.CodingRulesCodeMetrics.EnableCertC = true;`

`EnableCertCpp` — check CERT C++ rules
false (default) | true

This property affects Bug Finder only.

Check CERT C++ rules, specified as true or false. To customize which rules are checked, use `CertCpp`.

For more information about the CERT C++ checker, see `Check CERT-C++ security checks (-cert-cpp)`.

Example: `opts.CodingRulesCodeMetrics.EnableCertCpp = true;`

`EnableCheckersSelectionByFile` — Check custom set of coding standard checkers
false (default) | true

Check custom set of coding standard checkers, specified as true or false. Use with `CheckersSelectionByFile` and these coding standards:

- `opts.CodingRulesCodeMetrics.AutosarCpp14='from-file'`
- `opts.CodingRulesCodeMetrics.CertC='from-file'`
- `opts.CodingRulesCodeMetrics.CertCpp='from-file'`
- `opts.CodingRulesCodeMetrics.Iso17961='from-file'`
- `opts.CodingRulesCodeMetrics.JsfSubset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraC3Subset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraCSubset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraCppSubset='from-file'`

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCheckersSelectionByFile = true;`

`EnableCustomRules` — Check custom coding rules
false (default) | true

Check custom coding rules, specified as true or false. The file you specify with `CheckersSelectionByFile` defines the custom coding rules.

Use with `EnableCheckersSelectionByFile`.

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCustomRules = true;`

`EnableIso17961` — check ISO-17961 rules
false (default) | true

This property affects Bug Finder only.

Check ISO/IEC TS 17961 rules, specified as true or false. To customize which rules are checked, use `Iso17961`.

For more information about the ISO-17961 checker, see `Check ISO-17961 security checks (-iso-17961)`.

Example: `opts.CodingRulesCodeMetrics.EnableIso17961 = true;`

`EnableJsf` — Check JSF C++ rules
false (default) | true

Check JSF C++ rules, specified as true or false. To customize which rules are checked, use `JsfSubset`.

For more information, see `Check JSF C++ rules (-jsf-coding-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableJsf = true;`

`EnableMisraC` — Check MISRA C:2004 rules
false (default) | true

Check MISRA C:2004 rules, specified as true or false. To customize which rules are checked, use `MisraCSubset`.

For more information, see `Check MISRA C:2004 (-misra2)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraC = true;`

`EnableMisraC3` — Check MISRA C:2012 rules
false (default) | true

Check MISRA C:2012 rules, specified as true or false. To customize which rules are checked, use `MisraC3Subset`.

For more information about the MISRA C:2012 checker, see `Check MISRA C:2012 (-misra3)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraC3 = true;`

`EnableMisraCpp` — Check MISRA C++:2008 rules
false (default) | true

Check MISRA C++:2008 rules, specified as true or false. To customize which rules are checked, use `MisraCppSubset`.

For more information about the MISRA C++:2008 checker, see `Check MISRA C++ rules (-misra-cpp)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraCpp = true;`

`Iso17961` — Set of ISO-17961 rules to check
`'all'` (default) | `'decidable'` | `polyspace.CodingRulesOptions` object | `'from-file'`

This property affects Bug Finder only.

Set of ISO/IEC TS 17961 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check ISO-17961 security checks (-iso-17961)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check ISO/IEC TS 17961 rules, also set `EnableIso17961` to true.

Example: `opts.CodingRulesCodeMetrics.Iso17961 = 'all'`

Data Types: `char`

`JsfSubset` — Subset of JSF C++ rules to check
`'shall-rules'` (default) | `'shall-will-rules'` | `'all-rules'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of JSF C++ rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check JSF C++ rules (-jsf-coding-rules)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check JSF C++ rules, set `EnableJsf` to true.

Example: `opts.CodingRulesCodeMetrics.JsfSubset = 'all-rules'`

Data Types: `char`

`Misra3AgcMode` — Use the MISRA C:2012 categories for automatically generated code
false (default) | true

Use the MISRA C:2012 categories for automatically generated code, specified as true or false.

For more information, see `Use generated code requirements (-misra3-agc-mode)`.

Example: `opts.CodingRulesCodeMetrics.Misra3AgcMode = true;`

`MisraC3Subset` — Subset of MISRA C:2012 rules to check
`'mandatory-required'` (default) | `'mandatory'` | `'single-unit-rules'` | `'system-decidable-rules'` | `'all'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C:2012 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2012 (-misra3)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C:2012 rules, also set `EnableMisraC3` to true.

Example: `opts.CodingRulesCodeMetrics.MisraC3Subset = 'all'`

Data Types: `char`

`MisraCSubset` — Subset of MISRA C:2004 rules to check
`'required-rules'` (default) | `'single-unit-rules'` | `'system-decidable-rules'` | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C:2004 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2004 (-misra2)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created

automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C:2004 rules, also set `EnableMisraC` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCSubset = 'all-rules'`

Data Types: `char`

`MisraCppSubset` — Subset of MISRA C++ rules

`'required-rules'` (default) | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C++:2008 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C++ rules (-misra-cpp)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C++ rules, set `EnableMisraCpp` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCppSubset = 'all-rules'`

Data Types: `char`

EnvironmentSettings

`Dos` — Consider that file paths are in MS-DOS style

true (default) | false

Consider that file paths are in MS-DOS style, specified as true or false.

For more information, see `Code from DOS or Windows file system (-dos)`.

Example: `opts.EnvironmentSettings.Dos = true;`

`IncludeFolders` — Include folders needed for compilation

cell array of include folder paths

Include folders needed for compilation, specified as a cell array of the include folder paths.

To specify all subfolders of a folder, use folder path followed by **, for instance, `'C:\includes\**'`. The notation follows the syntax of the `dir` function. See also “Specify Multiple Source Files”.

For more information, see `-I`.

Example: `opts.EnvironmentSettings.IncludeFolders = {'/includes','/com1/inc'};`

Example: `opts.EnvironmentSettings.IncludeFolders = {'C:\project1\common \includes'};`

Data Types: `cell`

`Includes` — Files to be `#include`-ed by each C file
cell array of files

Files to be `#include`-ed by each C source file in the analysis, specified by a cell array of files.

For more information, see `Include (-include)`.

Example: `opts.EnvironmentSettings.Includes = {'/inc/inc_file.h','/inc/ inc_math.h'}`

`NoExternC` — Ignore linking errors inside extern blocks
false (default) | true

Ignore linking errors inside extern blocks, specified as true or false.

For more information, see `Ignore link errors (-no-extern-c)`.

Example: `opts.EnvironmentSettings.NoExternC = false;`

`PostPreProcessingCommand` — Command or script to run on source files after preprocessing
character vector

Command or script to run on source files after preprocessing, specified as a character vector of the command to run.

For more information, see `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

Example: Linux — `opts.EnvironmentSettings.PostPreProcessingCommand = [pwd,'/ replace_keyword.pl']`

Example: Windows — `opts.EnvironmentSettings.PostPreProcessingCommand = '"C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts \replace_keyword.pl"'`

`StopWithCompileError` — Stop analysis if a file does not compile
false (default) | true

Stop analysis if a file does not compile, specified as true or false.

For more information, see `Stop analysis if a file does not compile (-stop-if-compile-error)`.

Example: `opts.EnvironmentSettings.StopWithCompileError = true;`

`InputsStubbing`

`DataRangeSpecifications` — Constrain global variables, function inputs, and return values of stubbed functions
file path

Constrain global variables, function inputs, and return values of stubbed functions specified by the path to an XML constraint file. For more information about the constraint file, see “Specify External Constraints”.

For more information about this option, see `Constraint setup (-data-range-specifications)`.

Example: `opts.InputsStubbing.DataRangeSpecifications = 'C:\project\constraint_file.xml'`

`DoNotGenerateResultsFor` — Files on which you do not want analysis results
`'include-folders'` (default) | `'all-headers'` | `'custom=`*`folder1[,folder2,...]`*`'`

Files on which you do not want analysis results, specified by `'include-folders'`, `'all-headers'`, or a character array beginning with `custom=` followed by a comma-separated list of file or folder names.

Use this option with `InputsStubbing.GenerateResultsFor`. For more information, see `Do not generate results for (-do-not-generate-results-for)`.

Example: `opts.InputsStubbing.DoNotGenerateResultsFor = 'custom=C:\project\file1.c,C:\project\file2.c'`

`GenerateResultsFor` — Files on which you want analysis results
`'source-headers'` (default) | `'all-headers'` | `'custom=`*`folder1[,folder2,...]`*`'`

Files on which you want analysis results, specified by `'source-headers'`, `'all-headers'`, or a character array beginning with `custom=` followed by a comma-separated list of file or folder names.

Use this option with `InputsStubbing.DoNotGenerateResultsFor`. For more information, see `Generate results for sources and (-generate-results-for)`.

Example: `opts.InputsStubbing.GenerateResultsFor = 'custom=C:\project\includes_common_1,C:\project\includes_common_2'`

`FunctionsToStub` — Functions to stub during analysis
cell array of function names

This property affects Code Prover analysis only.

Functions to stub during analysis, specified as a cell array of function names.

For more information, see `Functions to stub (-functions-to-stub)`.

Example: `opts.InputsStubbing.FunctionsToStub = {'func1', 'func2'}`

`NoDefInitGlob` — Consider global variables as uninitialized
false (default) | true

This property affects Code Prover analysis only.

Consider global variables as uninitialized, specified as true or false.

For more information, see `Ignore default initialization of global variables (-no-def-init-glob)`.

Example: `opts.InputsStubbing.NoDefInitGlob = true`

`NoStlStubs` — Do not use Polyspace implementations of functions in the Standard Template Library
false (default) | true

This property applies only to a Code Prover analysis of C++ code.

Do not use Polyspace implementations of functions in the Standard Template Library, specified as true or false.

For more information, see `No STL stubs (-no-stl-stubs)`.

Example: `opts.InputsStubbing.NoStlStubs = true`

`StubECoderLookupTables` — Specify that the analysis must stub functions in the generated code that use lookup tables
true (default) | false

This property applies only to a Code Prover analysis of code generated from models.

Specify that the analysis must stub functions in the generated code that use lookup tables. By replacing the functions with stubs, the analysis assumes more precise return values for the functions.

For more information, see `Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions)`.

Example: `opts.InputsStubbing.StubECoderLookupTables = true`

Macros

`DefinedMacros` — Macros to be replaced
cell array of macros

In preprocessed code, macros are replaced by the definition, specified in a cell array of macros and definitions. Specify the macro as `Macro=Value`. If you want Polyspace to ignore the macro, leave the `Value` blank. A macro with no equal sign replaces all instances of that macro by 1.

For more information, see `Preprocessor definitions (-D)`.

Example: `opts.Macros.DefinedMacros = {'uint32=int','name3=','var'}`

`UndefinedMacros` — Macros to undefine
cell array of macros

In preprocessed code, macros are undefined, specified by a cell array of macros to undefine.

For more information, see `Disabled preprocessor definitions (-U)`.

Example: `opts.Macros.DefinedMacros = {'name1','name2'}`

MergedComputingSettings

`AddToResultsRepositoryBugFinder` — Upload Bug Finder results to Polyspace Metrics web dashboard
false (default) | true

This property affects Bug Finder analysis only.

Upload Bug Finder analysis results to Polyspace Metrics web dashboard, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Upload results to Polyspace Metrics (-add-to-results-repository)`.

Example: `opts.MergedComputingSettings.AddToResultsRepositoryBugFinder = true;`

`AddToResultsRepositoryCodeProver` — Upload Code Prover results to Polyspace Metrics web dashboard
false (default) | true

This property affects Code Prover analysis only.

Upload Code Prover analysis results to Polyspace Metrics web dashboard, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Upload results to Polyspace Metrics (-add-to-results-repository)`.

Example: `opts.MergedComputingSettings.AddToResultsRepositoryCodeProver = true;`

`BatchBugFinder` — Send Bug Finder analysis to remote server
false (default) | true

This property affects Bug Finder analysis only.

Send Bug Finder analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`.

Example: `opts.MergedComputingSettings.BatchBugFinder = true;`

`BatchCodeProver` — Send Code Prover analysis to remote server
false (default) | true

This property affects Code Prover analysis only.

Send Code Prover analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`.

Example: `opts.MergedComputingSettings.BatchCodeProver = true;`

`FastAnalysis` — Run Bug Finder analysis using faster local mode
false (default) | true

This property affects Bug Finder analysis only.

Use fast analysis mode for Bug Finder analysis, specified as true or false.

For more information, see `Use fast analysis mode for Bug Finder (-fast-analysis)`.

Example: `opts.MergedComputingSettings.FastAnalysis = true;`

MergedReporting

`EnableReportGeneration` — Generate a report after the analysis
false (default) | true

After the analysis, generate a report, specified as true or false.

For more information, see `Generate report`.

Example: `opts.MergedReporting.EnableReportGeneration = true`

`ReportOutputFormat` — Output format of generated report
`'Word'` (default) | `'HTML'` | `'PDF'`

Output format of generated report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Output format (-report-output-format)`.

Example: `opts.MergedReporting.ReportOutputFormat = 'PDF'`

`BugFinderReportTemplate` — Template for generating Bug Finder analysis report
`'BugFinderSummary'` (default) | `'BugFinder'` | `'SecurityCWE'` | `'CodeMetrics'` | `'CodingStandards'`

This property affects a Bug Finder analysis only.

Template for generating analysis report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Bug Finder and Code Prover report (-report-template)`.

Example: `opts.MergedReporting.BugFinderReportTemplate = 'CodeMetrics'`

`CodeProverReportTemplate` — Template for generating Code Prover analysis report
`'Developer'` (default) | `'CallHierarchy'` | `'CodeMetrics'` | `'CodingStandards'` | `'DeveloperReview'` | `'Developer_withGreenChecks'` | `'Quality'` | `'VariableAccess'`

This property affects a Code Prover analysis only.

Template for generating analysis report, specified as one of the predefined report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Bug Finder and Code Prover report (-report-template)`.

Example: `opts.MergedReporting.CodeProverReportTemplate = 'CodeMetrics'`

Multitasking

`ArxmlMultitasking` — Specify path of ARXML files to parse for multitasking configuration
cell array of file paths

Specify the path to the ARXML files the software parses to set up your multitasking configuration.

To activate this option, specify `Multitasking.EnableExternalMultitasking` and set `Multitasking.ExternalMultitaskingType` to `autosar`.

For more information, see `ARXML files selection (-autosar-multitasking)`

Example: `opts.Multitasking.ArxmlMultitasking={'C:\Polyspace_Workspace\AUTOSAR\myFile.arxml'}`

`CriticalSectionBegin` — Functions that begin critical sections
cell array of critical section function names

Functions that begin critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionEnd`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

Example: `opts.Multitasking.CriticalSectionBegin = {'function1:cs1','function2:cs2'}`

`CriticalSectionEnd` — Functions that end critical sections
cell array of critical section function names

Functions that end critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionBegin`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

Example: `opts.Multitasking.CriticalSectionEnd = {'function1:cs1','function2:cs2'}`

`CyclicTasks` — Specify functions that represent cyclic tasks
cell array of function names

Specify functions that represent cyclic tasks.

To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Cyclic tasks (-cyclic-tasks)`.

Example: `opts.Multitasking.CyclicTasks = {'function1','function2'}`

`EnableConcurrencyDetection` — Enable automatic detection of certain families of threading functions
false (default) | true

This property affects Code Prover analysis only.

Enable automatic detection of certain families of threading functions, specified as true or false.

For more information, see `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

Example: `opts.Multitasking.EnableConcurrencyDetection = true`

`EnableExternalMultitasking` — Enable automatic multitasking configuration from external file definitions
false (default) | true

Enable multitasking configuration of your projects from external files you provide. Configure multitasking from ARXML files for an AUTOSAR project, or from OIL files for an OSEK project.

Activate this option to enable `Multitasking.ArxmlMultitasking` or `Multitasking.OsekMultitasking`.

For more information, see `OIL files selection (-osek-multitasking)` and `ARXML files selection (-autosar-multitasking)`.

Example: `opts.Multitasking.EnableExternalMultitasking = 1`

`EnableMultitasking` — Configure multitasking manually
false (default) | true

Configure multitasking manually by specifying `true`. This property activates the other manual, multitasking properties.

For more information, see `Configure multitasking manually`.

Example: `opts.Multitasking.EnableMultitasking = 1`

`EntryPoints` — Functions that serve as entry-points to your multitasking application
cell array of entry-point function names

Functions that serve as entry-points to your multitasking application specified as a cell array of entry-point function names. To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Tasks (-entry-points)`.

Example: `opts.Multitasking.EntryPoints = {'function1','function2'}`

`ExternalMultitaskingType` — Specify type of file to parse for multitasking configuration
`'osek'` (default) | `'autosar'`

Specify the type of file the software parses to set up your multitasking configuration:

- For `osek` type, the analysis looks for OIL files in the file or folder paths that you specify.
- For `autosar` type, the analysis looks for ARXML files in the file paths that you specify.

To activate this option, specify `Multitasking.EnableExternalMultitasking`.

For more information, see `OIL files selection (-osek-multitasking)` and `ARXML files selection (-autosar-multitasking)`.

Example: `opts.Multitasking.ExternalMultitaskingType = 'autosar'`

`Interrupts` — Specify functions that represent nonpreemptable interrupts
cell array of function names

Specify functions that represent nonpreemptable interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Interrupts (-interrupts)`.

Example: `opts.Multitasking.Interrupts = {'function1','function2'}`

`InterruptsDisableAll` — Specify routine that disable interrupts
cell array with one function name

This property affects Bug Finder analysis only.

Specify function that disables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Example: `opts.Multitasking.InterruptsDisableAll = {'function'}`

`InterruptsEnableAll` — Specify routine that reenable interrupts
cell array with one function name

This property affects Bug Finder analysis only.

Specify function that reenables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Example: `opts.Multitasking.InterruptsEnableAll = {'function'}`

`OsekMultitasking` — Specify path of OIL files to parse for multitasking configuration
`'auto'` (default) | `'custom=folder1[,folder2,...]'`

Specify the path to the OIL files the software parses to set up your multitasking configuration:

- In the mode specified with `'auto'`, the analysis uses OIL files in your project source and include folders, but not their subfolders.
- In the mode specified with `'custom=folder1[,folder2,...]'`, the analysis uses the OIL files at the specified path, and the path subfolders.

To activate this option, specify `Multitasking.EnableExternalMultitasking` and set `Multitasking.ExternalMultitaskingType` to `osek`.

For more information, see `OIL files selection (-osek-multitasking)`

Example: `opts.Multitasking.OsekMultitasking = 'custom=file_path, dir_path'`

`TemporalExclusion` — Entry-point functions that cannot execute concurrently
cell array of entry-point function names

Entry-point functions that cannot execute concurrently specified as a cell array of entry-point function names. Each set of exclusive tasks is one cell array entry with functions separated by spaces. To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Temporally exclusive tasks (-temporal-exclusions-file)`.

Example: `opts.Multitasking.TemporalExclusion = {'function1 function2', 'function3 function4 function5'}` where function1 and function2 are temporally exclusive, and function3, function4, and function 5 are temporally exclusive.

Precision (Affects Code Prover Only)

`ContextSensitivity` — Store call context information to identify function call that caused errors
`'none'` (default) | `'auto'` | `'custom=`*`function1[,function2,...]`*`'`

This property affects Code Prover analysis only.

Store call context information to identify a function call that caused errors, specified as `none`, `auto`, or as a character array beginning with `custom=` followed by a list of comma-separated function names.

For more information, see `Sensitivity context (-context-sensitivity)`.

Example: `opts.Precision.ContextSensitivity = 'auto'`

Example: `opts.Precision.ContextSensitivity = 'custom=func1'`

`ModulesPrecision` — Source files you want to verify at higher precision
cell array of file names and precision levels

This property affects Code Prover analysis only.

Source files that you want to verify at higher precision, specified as a cell array of file names without the extension and precision levels using this syntax: *`filename`*`:O`*`level`*

For more information, see `Specific precision (-modules-precision)`.

Example: `opts.Precision.ModulesPrecision = {'file1:O0', 'file2:O3'}`

`OLevel` — Precision level for the verification
2 (default) | 0 | 1 | 3

This property affects Code Prover analysis only.

Precision level for the verification, specified as 0, 1, 2, or 3.

For more information, see `Precision level (-O)`.

Example: `opts.Precision.OLevel = 3`

`PathSensitivityDelta` — Avoid certain verification approximations for code with fewer lines
positive integer

This property affects Code Prover analysis only.

Avoid certain verification approximations for code with fewer lines, specified as a positive integer representing how sensitive the analysis is. Higher values can increase verification time exponentially.

For more information, see `Improve precision of interprocedural analysis (-path-sensitivity-delta)`.

Example: `opts.Precision.PathSensitivityDelta = 2`

`Timeout` — Time limit on your verification
character vector

This property affects Code Prover analysis only.

Time limit on your verification, specified as a character vector of time in hours.

For more information, see `Verification time limit (-timeout)`.

Example: `opts.Precision.Timeout = '5.75'`

`To` — Number of times the verification process runs

`'Software Safety Analysis level 2'` (default) | `'Software Safety Analysis level 0'` | `'Software Safety Analysis level 1'` | `'Software Safety Analysis level 3'` | `'Software Safety Analysis level 4'` | `'Source Compliance Checking'` | `'other'`

This property affects Code Prover analysis only.

Number of times the verification process runs, specified as one of the preset analysis levels.

For more information, see `Verification level (-to)`.

Example: `opts.Precision.To = 'Software Safety Analysis level 3'`

Scaling (Affects Code Prover Only)

`Inline` — Functions on which separate results must be generated for each function call

cell array of function names

This property affects Code Prover analysis only.

Functions on which separate results must be generated for each function call, specified as a cell array of function names.

For more information, see `Inline (-inline)`.

Example: `opts.Scaling.Inline = {'func1','func2'}`

`KLimiting` — Limit depth of analysis for nested structures

positive integer

This property affects Code Prover analysis only.

Limit depth of analysis for nested structures, specified as a positive integer indicating how many levels into a nested structure to verify.

For more information, see `Depth of verification inside structures (-k-limiting)`.

Example: `opts.Scaling.KLimiting = 3`

TargetCompiler

`Compiler` — Compiler that builds your source code

`'generic'` (default) | `'gnu3.4'` | `'gnu4.6'` | `'gnu4.7'` | `'gnu4.8'` | `'gnu4.9'` | `'gnu5.x'` | `'gnu6.x'` | `'gnu7.x'` | `'clang3.x'` | `'clang4.x'` | `'clang5.x'` | `'visual9.0'` | `'visual10'` | `'visual11.0'` | `'visual12.0'` | `'visual14.0'` | `'visual15.x'` | `'keil'` | `'iar'` | `'armcc'` | `'armclang'` | `'codewarrior'` | `'diab'` | `'greenhills'` | `'iar-ew'` | `'renesas'` | `'tasking'` | `'ti'`

Compiler that builds your source code.

For more information, see `Compiler (-compiler)`.

Example: `opts.TargetCompiler.Compiler = 'Visual11.0'`

`CppVersion` — Specify C++11 standard version followed in code
`'defined-by-compiler'` (default) | `'cpp03'` | `'cpp11'` | `'cpp14'`

Specify C++ standard version followed in code, specified as a character vector.

For more information, see `C++ standard version (-cpp-version)`.

Example: `opts.TargetCompiler.CppVersion = 'cpp11';`

`CVersion` — Specify C standard version followed in code
`'defined-by-compiler'` (default) | `'c90'` | `'c99'` | `'c11'`

Specify C standard version followed in code, specified as a character vector.

For more information, see `C standard version (-c-version)`.

Example: `opts.TargetCompiler.CVersion = 'c90';`

`DivRoundDown` — Round down quotients from division or modulus of negative numbers
false (default) | true

Round down quotients from division or modulus of negative numbers, specified as true or false.

For more information, see `Division round down (-div-round-down)`.

Example: `opts.TargetCompiler.DivRoundDown = true`

`EnumTypeDefinition` — Base type representation of enum
`'defined-by-compiler'` (default) | `'auto-signed-first'` | `'auto-unsigned-first'`

Base type representation of enum, specified by an allowed base-type set. For more information about the different values, see `Enum type definition (-enum-type-definition)`.

Example: `opts.TargetCompiler.EnumTypeDefinition = 'auto-unsigned-first'`

`IgnorePragmaPack` — Ignore #pragma pack directives
false (default) | true

Ignore #pragma pack directives, specified as true or false.

For more information, see `Ignore pragma pack directives (-ignore-pragma-pack)`.

Example: `opts.TargetCompiler.IgnorePragmaPack = true`

`Language` — Language of analysis
`'C-CPP'` (default) | `'C'` | `'CPP'`

This property is read-only.

Language of the analysis, specified during the object construction. This value changes which properties appear.

For more information, see `Source code language (-lang)`.

`LogicalSignedRightShift` — Treatment of signed bit on signed variables
`'Arithmetical'` (default) | `'Logical'`

Treatment of signed bit on signed variables, specified as `Arithmetical` or `Logical`. For more information, see `Signed right shift (-logical-signed-right-shift)`.

Example: `opts.TargetCompiler.LogicalSignedRightShift = 'Logical'`

`NoUliterals` — Do not use predefined typedefs for char16_t or char32_t
false (default) | true

Do not use predefined typedefs for char16_t or char32_t, specified as true or false. For more information, see `Block char16/32_t types (-no-uliterals)`.

Example: `opts.TargetCompiler.NoUliterals = true`

`PackAlignmentValue` — Default structure packing alignment
`'defined-by-compiler'` (default) | `'1'` | `'2'` | `'4'` | `'8'` | `'16'`

Default structure packing alignment, specified as `'defined-by-compiler'`, `'1'`, `'2'`, `'4'`, `'8'`, or `'16'`. This property is available only for Visual C++ code.

For more information, see `Pack alignment value (-pack-alignment-value)`.

Example: `opts.TargetCompiler.PackAlignmentValue = '4'`

`SfrTypes` — sfr types
cell array of `sfr` keywords

`sfr` types, specified as a cell array of `sfr` keywords using the syntax *`sfr_name=size_in_bits`*. For more information, see `Sfr type support (-sfr-types)`.

This option only applies when you set `TargetCompiler.Compiler` to `keil` or `iar`.

Example: `opts.TargetCompiler.SfrTypes = {'sfr32=32'}`

`SizeTTypeIs` — Underlying type of `size_t`
`'defined-by-compiler'` (default) | `'unsigned-int'` | `'unsigned-long'` | `'unsigned-long-long'`

Underlying type of `size_t`, specified as `'defined-by-compiler'`, `'unsigned-int'`, `'unsigned-long'`, or `'unsigned-long-long'`. See `Management of size_t (-size-t-type-is)`.

Example: `opts.TargetCompiler.SizeTTypeIs = 'unsigned-long'`

`Target` — Target processor
`'i386'` (default) | `'arm'` | `'arm64'` | `'avr'` | `'c-167'` | `'c166'` | `'c18'` | `'c28x'` | `'c6000'` | `'coldfire'` | `'hc08'` | `'hc12'` | `'m68k'` | `'mcore'` | `'mips'` | `'mpc5xx'` | `'msp430'` | `'necv850'` | `'powerpc'` | `'powerpc64'` | `'rh850'` | `'rl78'` | `'rx'` | `'s12z'` | `'sharc21x61'` | `'sparc'` | `'superh'` | `'tms320c3x'` | `'tricore'` | `'x86_64'` | generic target object

Set size of data types and endianness of processor, specified as one of the predefined target processors or a generic target object.

For more information about the predefined processors, see `Target processor type (-target)`.

For more information about creating a generic target, see `polyspace.GenericTargetOptions`.

Example: `opts.TargetCompiler.Target = 'hc12'`

`WcharTTypeIs` — Underlying type of `wchar_t`
`'defined-by-compiler'` (default) | `'signed-short'` | `'unsigned-short'` | `'signed-int'` | `'unsigned-int'` | `'signed-long'` | `'unsigned-long'`

Underlying type of `wchar_t`, specified as `'defined-by-compiler'`, `'signed-short'`, `'unsigned-short'`, `'signed-int'`, `'unsigned-int'`, `'signed-long'`, or `'unsigned-long'`. See `Management of wchar_t (-wchar-t-type-is)`.

Example: `opts.TargetCompiler.WcharTTypeIs = 'unsigned-int'`

VerificationAssumption (Affects Code Prover Only)

`ConsiderVolatileQualifierOnFields` — Assume that volatile qualified structure fields can have all possible values at any point in code
false (default) | true

This property affects Code Prover analysis only.

Assume that volatile qualified structure fields can have all possible values at any point in code.

For more information, see `Consider volatile qualifier on fields (-consider-volatile-qualifier-on-fields)`.

Example: `opts.VerificationAssumption.ConsiderVolatileQualifierOnFields = true`

`ConstraintPointersMayBeNull` — Specify that environment pointers can be NULL unless constrained otherwise
false (default) | true

This property affects Code Prover analysis only.

Specify that environment pointers can be NULL unless constrained otherwise.

For more information, see `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`.

Example: `opts.VerificationAssumption.ConstraintPointersMayBeNull = true`

`FloatRoundingMode` — Rounding modes to consider when determining the results of floating-point arithmetic
`to-nearest` (default) | `all`

This property affects Code Prover analysis only.

Rounding modes to consider when determining the results of floating-point arithmetic, specified as `to-nearest` or `all`.

For more information, see `Float rounding mode (-float-rounding-mode)`.

Example: `opts.VerificationAssumption.FloatRoundingMode = 'all'`

`RespectTypesInFields` — Do not cast nonpointer fields of a structure to pointers
false (default) | true

This property affects Code Prover analysis only.

Do not cast nonpointer fields of a structure to pointers, specified as true or false.

For more information, see `Respect types in fields (-respect-types-in-fields)`.

Example: `opts.VerificationAssumption.RespectTypesInFields = true`

`RespectTypesInGlobals` — Do not cast nonpointer global variables to pointers
false (default) | true

This property affects Code Prover analysis only.

Do not cast nonpointer global variables to pointers, specified as true or false.

For more information, see `Respect types in global variables (-respect-types-in-globals)`.

Example: `opts.VerificationAssumption.RespectTypesInGlobals = true`

Other Properties

`Author` — Project author
username of current user (default) | character vector

Name of project author, specified as a character vector.

For more information, see `-author`.

Example: `opts.Author = 'JaneDoe'`

`ImportComments` — Import comments and justifications from previous analysis
character vector

To import comments and justifications from a previous analysis, specify the path to the results folder of the previous analysis.

You can also point to a previous results folder to see only new results compared to the previous run. See “Compare Results from Different Polyspace Runs by Using MATLAB Scripts”.

For more information, see `-import-comments`

Example: `opts.ImportComments = fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example','Module_1','BF_Result')`

`Prog` — Project name
`PolyspaceProject` (default) | character vector

Project name, specified as a character vector.

For more information, see `-prog`.

Example: `opts.Prog = 'myProject'`

`ResultsDir` — Location to store results
folder path

Location to store results, specified as a folder path. By default, the results are stored in the current folder.

For more information, see `-results-dir`.

You can also create a separate results folder for each new run. See “Compare Results from Different Polyspace Runs by Using MATLAB Scripts”.

Example: `opts.ResultsDir = 'C:\project\myproject\results\'`

`Sources` — Source files
cell array of files

Source files to analyze, specified as a cell array of files.

To specify all files in a folder, use folder path followed by *, for instance, `'C:\src\*'`. To specify all files in a folder and its subfolders, use folder path followed by **, for instance, `'C:\src\**'`. The notation follows the syntax of the `dir` function. See also “Specify Multiple Source Files”.

For more information, see `-sources`.

Example: `opts.Sources = {'file1.c', 'file2.c', 'file3.c'}`

Example: `opts.Sources = {'project/src1/file1.c', 'project/src2/file2.c', 'project/src3/file3.c'}`

`Version` — Project version number
`'1.0'` (default) | character array of a number

Version number of project, specified as a character array of a number. This option is useful if you upload your results to Polyspace Metrics. If you increment version numbers each time that you reanalyze your object, you can compare the results from two versions in Polyspace Metrics.

For more information, see `-v[ersion]`.

Example: `opts.Version = '2.3'`

See Also

Topics
“Analysis Options”

Introduced in R2017a

copyTo

Class: `polyspace.Options`
Package: `polyspace`

Copy common settings between Polyspace options objects

Syntax

```
optsFrom.copyTo(optsTo)
```

Description

`optsFrom.copyTo(optsTo)` copies the common options from `optsFrom` to `optsTo`. The options objects do not need to be the same type of options object. This method copies only properties that are common between the two objects.

Input Arguments

`optsFrom` — Options object you want to copy properties from

`polyspace.Options` or `polyspace.ModelLinkOptions` object

Option object that you want to copy properties from, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

`optsTo` — Options object you want to copy properties to

`polyspace.Options` object

Option object that you want to copy properties to, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

Examples

Copy Polyspace Options Object

This example shows how to set the properties of one options object and then copy that object to another one.

Create a Polyspace options object and set properties.

```
opts1 = polyspace.Options();
opts1.Prog = 'DataRaceProject';
opts1.Sources = {'datarace.c'};
opts1.TargetCompiler.Compiler = 'gnu4.9';
```

Create another object and use copyTo to copy over options from the previous object.

```
opts2 = polyspace.Options();
opts1.copyTo(opts2);
```

See Also

generateProject | polyspace.ModelLinkOptions | polyspace.Options

Introduced in R2016b

generateProject

Class: `polyspace.Options`
Package: `polyspace`

Generate psprj project from options object

Syntax

```
opts.generateProject(projectName)
```

Description

`opts.generateProject(projectName)` creates a `.psprj` project called `projectName` from the options specified in the `polyspace.Options` object `opts`. You can open a `.psprj` project in the user interface of the Polyspace desktop products.

Input Arguments

`opts` — Options object to convert into a `psprj` file

`polyspace.Options` or `polyspace.ModelLinkOptions` object

Option object convert into a `psprj` file, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

`projectName` — Project file name

character vector

Project file name specified as a character vector. This argument is used as the name of the `psprj` file.

Example: `'myProject'`

Examples

Generate Project from a Bug Finder Options Object

This example shows how to create and use a Polyspace project that was generated from an options object.

Create a Bug Finder object and set properties.

```
sources = fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example',...
   'sources','numerical.c');
opts = polyspace.Options();
opts.Prog = 'MyProject';
opts.Sources = {sources};
opts.TargetCompiler.Compiler = 'gnu4.7';
```

Generate a Polyspace project. Name the project using the `Prog` property.

```
psprj = opts.generateProject(opts.Prog);
```

Run a Bug Finder analysis using one of these commands. Both commands produce identical analysis results. The only difference is that the `psprj` project can be rerun in the Polyspace interface.

```
polyspaceBugFinder(psprj, '-nodesktop');
polyspaceBugFinder(opts);
```

To run a Code Prover analysis, use `polyspaceCodeProver` instead of `polyspaceBugFinder`.

Tips

If you want to include an options object in a `pslinkoptions` object:

1. Use this method to convert your object to a project.
2. Add the project to the `pslinkoptions` property `PrjConfig`.
3. Turn on the property `EnablePrjConfig`.

See Also

`copyTo` | `polyspace.ModelLinkOptions` | `polyspace.Options`

Introduced in R2016b

toScript

Class: `polyspace.Options`
Package: `polyspace`

Add Polyspace options object definition to a script

Syntax

```
filePath = opts.toScript(fileName,positionInScript)
```

Description

`filePath = opts.toScript(fileName,positionInScript)` adds the properties of a `polyspace.Options` object to a MATLAB script. The script shows the values assigned to all the properties of the object. You can run the script later to define the object in the MATLAB workspace and use it.

Input Arguments

`opts` — Options object with Polyspace analysis options

`polyspace.Options` or `polyspace.ModelLinkOptions` object

Option object to store in MATLAB script, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

`fileName` — Script name

character vector

Name or path to script, specified as a character vector. If you specify a relative path, the script is created in subfolder of the current working folder.

Example: `'runPolyspace.m'`

`positionInScript` — Where to add object definition

`'create'` (default) | `'append'`

Position in script where the object properties are added, specified as `'create'` or `'append'`. If you specify `'append'`, the object properties are added to the end of an existing script. Otherwise, a new script is created.

Output Arguments

`filePath` — Full path to script

character vector

Full path to script, specified as a character vector.

Example: `'C:\myScripts\runPolyspace.m'`

See Also

`copyTo` | `generateProject` | `polyspace.ModelLinkOptions` | `polyspace.Options`

Introduced in R2017b

run

Run a Polyspace analysis

Syntax

```
run(proj, product)
```

Description

`status = run(proj, product)` runs a Polyspace Bug Finder or Polyspace Code Prover analysis using the configuration specified in the `polyspace.Project` object `proj`. The analysis results are also stored in `proj`.

Input Arguments

`proj` — Polyspace project

`polyspace.Project` object

Polyspace project with configuration and results, specified as a `polyspace.Project` object.

`product` — Type of analysis

`'bugFinder'` | `'codeProver'`

Type of analysis to run.

Output Arguments

`status` — Results of a Code Prover analysis

`true` | `false`

Status of analysis. If the analysis fails, the status is `false`. Otherwise, it is `true`.

The analysis can fail for multiple reasons:

- You provide source files that do not exist.
- None of your files compile. Even if one file compiles, unless you set the property `StopWithCompileError` to `true`, the analysis succeeds and returns a `true` status.

There can be many other reasons why the analysis fails. If the analysis fails, in your results folder, check the log file. You can see the results folder using the `Configuration` property of the `polyspace.Project` object:

```
proj = polyspace.Project;
proj.Configuration.ResultsDir
```

The log file is named `Polyspace_R20##n_ProjectName_date-time.log`.

Examples

Read Results to MATLAB Tables

Run a Polyspace Bug Finder analysis on the demo file `numerical.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = run(proj, 'bugFinder');

% Read results
bfSummary = proj.Results.getSummary('defects');
```

Introduced in R2017b

getSummary

Class: `polyspace.BugFinderResults`
Package: `polyspace`

View number of defects organized by defect type

Syntax

```
resObj.getSummary(resultsType)
```

Description

`resSummary = resObj.getSummary(resultsType)` returns the distribution of results of type `resultsType` in a Bug Finder result set denoted by the `polyspace.BugFinderResults` object `resObj`. For instance, if you choose to see defects, you can see how many defects of each type are present in the result set, for instance, how many non-initialized variables or declaration mismatches.

Input Arguments

`resultsType` — Type of Bug Finder analysis result

`'defects'` (default) | `'misraC'` | `'misraCAGC'` | `'misraCPP'` | `'misraC2012'` | `'jsf'` | `'metrics'` | `'customRules'`

Type of result, specified as a character vector.

Entry	Meaning
`'defects'`	Bugs or defects.
`'misraC'`	MISRA C:2004 rules.
`'misraCAGC'`	MISRA C:2004 rules for generated code.
`'misraCPP'`	MISRA C++ rules.
`'misraC2012'`	MISRA C:2012 rules.
`'jsf'`	JSF C++ rules.
`'metrics'`	Code complexity metrics.
`'customRules'`	Custom rules enforcing naming conventions for identifiers.

Output Arguments

`resSummary` — Distribution of defects by defect type

table

Distribution of defects by defect type, specified as a table. For instance, an extract of the table looks like this:

Category	Defect	Impact	Total
Concurrency	Data race	High	2
Concurrency	Deadlock	High	1
Data flow	Non-initialized variable	High	2

The table above shows that the result set contains two data races, one deadlock and two non-initialized variables.

For more information on MATLAB tables, see “Tables” (MATLAB).

Examples

Copy Existing Results to MATLAB Tables

This example shows how to read Bug Finder analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath=fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example', ...
'Module_1','BF_Result');
userResPath = tempname;
copyfile(resPath,userResPath);
```

Create the results object.

```
resObj = polyspace.BugFinderResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = resObj.getSummary('defects');
resTable = resObj.getResults();
```

Run Analysis and Read Results to MATLAB Tables

Run a Polyspace Bug Finder analysis on the demo file `numerical.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = proj.run('bugFinder');

% Read results
bfSummary = proj.Results.getSummary('defects');
```

See Also

`polyspace.BugFinderResults`

Topics
“Defects”
“Bug Finder Defect Groups”
“Classification of Defects by Impact”

Introduced in R2017a

getResults

Class: `polyspace.BugFinderResults`
Package: `polyspace`

Read Bug Finder results into MATLAB table

Syntax

```
resObj.getResults(content)
```

Description

`resTable = resObj.getResults(content)` returns a table showing all results in a Bug Finder result set denoted by the `polyspace.BugFinderResults` object `resObj`. You can manipulate the table to produce graphs and statistics about your results that you cannot obtain readily from the user interface.

Input Arguments

`content` — Result information to include

`''` (default) | `'readable'`

Amount of information to be included for each result. If you specify `''`, all information is included. If you specify `'readable'`, the following information is not included:

- ID: Unique number for a result for the current analysis.
- Group: Defect groups, MISRA C:2012 groups, etc.
- Status, Severity, Comment: Information that *you* enter about a result.

If you do not specify this argument, the full table is included.

See “Export Polyspace Analysis Results”.

Output Arguments

`resTable` — Results of a Bug Finder analysis

table

Table showing all results from a single Bug Finder analysis. For each result, the table has information such as file, family, and so on. If a particular information is not available for a result, the entry in the table states `<undefined>`.

For more information on:

- The columns of the table, see “Export Polyspace Analysis Results”.
- MATLAB tables, see “Tables” (MATLAB).

Examples

Copy Existing Results to MATLAB Tables

This example shows how to read Bug Finder analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath = fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example', ...
'Module_1','BF_Result');
userResPath = tempname;
copyfile(resPath,userResPath);
```

Create the results object.

```
resObj = polyspace.BugFinderResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = resObj.getSummary ('defects');
resTable = resObj.getResults ('');
```

Run Analysis and Read Results to MATLAB Tables

Run a Polyspace Bug Finder analysis on the demo file `numerical.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = proj.run('bugFinder');

% Read results
bfSummary = proj.Results.getResults('readable');
```

See Also

`polyspace.BugFinderResults`

Introduced in R2017a

MISRA C 2012

MISRA C:2012 Rule 1.1

The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits

Description

Rule Definition

The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.

Polyspace Implementation

The rule checker checks for these issues. The specifications can depend on the version of the C standard used in the analysis. See `C standard version (-c-version)`.

Issue	C Standard Dependence	Additonal Information
An integer constant falls outside the range of `long int` (if the constant is signed) or `unsigned long int` (if the constant is unsigned).	Checked for C90 only.	The rule checker uses your specifications for the size of a `long int` variable (typically 32 bits). See also `Target processor type (-target)`.
An array of size zero is used.	Checked for C90 only.	
The number of macros defined in a translation unit exceeds the limit specified in the standard.	Number of macro definitions allowed: • C90: 1024 • C99 and later: 4095	The rule checker considers a translation unit as a source file and header files included directly or indirectly in the source file.
The depth of nesting exceeds the limit specified in the standard.	Maximum nesting depth allowed: • C90: 15 • C99 and later: 127	
The number of levels of conditional inclusion (with `#if`, etc.) exceeds the limit specified in the standard.	Maximum number of levels of conditional inclusion allowed: • C90: 8 • C99 and later: 15	
The number of members of a structure or union exceeds the limit specified in the standard.	Maximum number of members in a structure or union: • C90: 127 (with 15 levels of nesting) • C99 and later: 1023 (with 63 levels of nesting)	

Issue	C Standard Dependence	Additonal Information
The number of constants in a single enumeration exceeds the limit specified in the standard.	Maximum number of enumeration constants allowed: • C90: 127 • C99 and later: 1023	
An asssembly language statement is used.	Checked for all C standards.	
A nonstandard preprocessor directive is used.	Checked for all C standards.	The rule checker flags uses of preprocessor directives that are not found in the C standard, for instance, `#ident`, `#alias` and `#assert`.
Unrecognized text follows a preprocessor directive.	Checked for all C standards.	The rule checker flags extraneous text following a preprocessor directive (line beginning with #). For instance: `#include <header> code`

Standard compilation error messages do not lead to a violation of this MISRA rule.

Tip To mass-justify all results that come from the same cause, use the **Detail** column on the **Results List** pane. Click the column header so that all results with the same entry are grouped together. Select the first result and then select the last result while holding the `Shift` key. Assign a status to one of the results. If you do not see the **Detail** column, right-click any other column header and enable this column.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Standard C Environment
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 1.2` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 1.2

Language extensions should not be used

Description

Rule Definition

Language extensions should not be used.

Rationale

If a program uses language extensions, its portability is reduced. Even if you document the language extensions, the documentation might not describe the behavior in all circumstances.

Polyspace Implementation

The rule checker flags these language extensions, depending on the version of the C standard used in the analysis. See `C standard version (-c-version)`.

- C90:
  - `long long int` type including constants
  - `long double` type
  - `inline` keyword
  - `_Bool` keyword
  - `short long int` type
  - Hexadecimal floating-point constants
  - Universal character names
  - Designated initializers
  - Local label declarations
  - `typeof` operator
  - Casts to union
  - Compound literals
  - Statements and declarations in expressions
  - `__func__` predefined identifier
  - `_Pragma` preprocessing operator
  - Macros with variable arguments list
- C99:
  - `short long int` type
  - Local label declarations
  - `typeof` operator
  - Casts to union
  - Statements and declarations in expressions

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Standard C Environment
Category: Advisory
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 1.1` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 1.3

There shall be no occurrence of undefined or critical unspecified behaviour

Description

Rule Definition

There shall be no occurrence of undefined or critical unspecified behaviour.

Additional Message in Report

There shall be no occurrence of undefined or critical unspecified behavior

- 'defined' without an identifier.
- macro 'XX' used with too few arguments.
- macro 'XX used with too many arguments.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Standard C Environment
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Dir 4.1|Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 10.1

Operands shall not be of an inappropriate essential type

Description

Rule Definition

Operands shall not be of an inappropriate essential type.

Rationale

What Are Essential Types?

An essential type category defines the essential type of an object or expression.

Essential type category	Standard types
Essentially Boolean	`bool` or `_Bool` (defined in `stdbool.h`) You can also define types that are essentially Boolean using the option `Effective boolean types (-boolean-types)`.
Essentially character	`char`
Essentially enum	named `enum`
Essentially signed	signed `char`, signed `short`, signed `int`, signed `long`, signed `long long`
Essentially unsigned	unsigned `char`, unsigned `short`, unsigned `int`, unsigned `long`, unsigned `long long`
Essentially floating	`float`, `double`, `long double`

Amplification and Rationale

For operands of some operators, you cannot use certain essential types. In the table below, each row represents an operator/operand combination. If the essential type column is not empty for that row, there is a MISRA restriction when using that type as the operand. The number in the table corresponds to the rationale list after the table.

Operation		Essential type category of arithmetic operand					
Operator	**Operand**	**Boolean**	**character**	**enum**	**signed**	**unsigned**	**floating**
`[ ]`	integer	3	4				1
+ (unary)		3	4	5			
- (unary)		3	4	5		8	
`+ -`	either	3		5			
`* /`	either	3	4	5			
`%`	either	3	4	5			1
`< > <= >=`	either	3					

Operation		Essential type category of arithmetic operand					
== !=	either						
! && ||	any		2	2	2	2	2
<< >>	left	3	4	5,6	6		1
<< >>	right	3	4	7	7		1
~ & | ^	any	3	4	5,6	6		1
?:	1st		2	2	2	2	2
?:	2nd and 3rd						

1. An expression of essentially floating type for these operands is a constraint violation.
2. When an operand is interpreted as a Boolean value, use an expression of essentially Boolean type.
3. When an operand is interpreted as a numeric value, do not use an operand of essentially Boolean type.
4. When an operand is interpreted as a numeric value, do not use an operand of essentially character type. The numeric values of character data are implementation-defined.
5. In an arithmetic operation, do not use an operand of essentially enum type. An enum object uses an implementation-defined integer type. An operation involving an enum object can therefore yield a result with an unexpected type.
6. Perform only shift and bitwise operations on operands of essentially unsigned type. When you use shift and bitwise operations on essentially signed types, the resulting numeric value is implementation-defined.
7. To avoid undefined behavior on negative shifts, use an essentially unsigned right-hand operand.
8. For the unary minus operator, do not use an operand of essentially unsigned type. The implemented size of int determines the signedness of the result.

Additional Message in Report

The *`operand_name`* operand of the *`operator_name`* operator is of an inappropriate essential type category *`category_name`*.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Violation of Rule 10.1, Rationale 2: Inappropriate Operand Types for Operators That Take Essentially Boolean Operands

```
typedef unsigned char boolean;

extern float f32a;
extern char cha;
extern signed char s8a;
extern unsigned char u8a;
enum enuma { a1, a2, a3 } ena;

extern boolean bla, blb, rbla;

void foo(void) {
```

```
  rbla = cha && bla;        /* Non-compliant: cha is essentially char  */
  enb = ena ? a1 : a2;      /* Non-compliant: ena is essentially enum  */
  rbla = s8a && bla;        /* Non-compliant: s8a is essentially signed char  */
  ena = u8a ? a1 : a2;      /* Non-compliant: u8a is essentially unsigned char  */
  rbla = f32a && bla;       /* Non-compliant: f32a is essentially float */

  rbla = bla && blb;        /* Compliant */
  ru8a = bla ? u8a : u8b;   /* Compliant */

}
```

In the noncompliant examples, rule 10.1 is violated because:

- The operator && expects only essentially Boolean operands. However, at least one of the operands used has a different type.
- The first operand of `?:` is expected to be essentially Boolean. However, a different operand type is used.

Note For Polyspace to detect the rule violation, you must define the type name `boolean` as an effective Boolean type. For more information, see `Effective boolean types (-boolean-types)`.

Violation of Rule 10.1, Rationale 3: Inappropriate Boolean Operands

```
typedef unsigned char boolean;


enum enuma { a1, a2, a3 } ena;
enum { K1 = 1, K2 = 2 };    /* Essentially signed */
extern char cha, chb;
extern boolean bla, blb, rbla;
extern signed char rs8a, s8a;


void foo(void) {

  rbla = bla * blb;      /* Non-compliant - Boolean used as a numeric value */
  rbla = bla > blb;      /* Non-compliant - Boolean used as a numeric value */

  rbla = bla && blb;     /* Compliant */
  rbla = cha > chb;      /* Compliant */
  rbla = ena > a1;       /* Compliant */
  rbla = u8a > 0U;       /* Compliant */
  rs8a = K1 * s8a;       /* Compliant - K1 obtained from anonymous enum */

}
```

In the noncompliant examples, rule 10.1 is violated because the operators * and > do not expect essentially Boolean operands. However, the operands used here are essentially Boolean.

Note For Polyspace to detect the rule violation, you must define the type name `boolean` as an effective Boolean type. For more information, see `Effective boolean types (-boolean-types)`.

Violation of Rule 10.1, Rationale 4: Inappropriate Character Operands

```
extern char rcha, cha, chb;
extern unsigned char ru8a, u8a;

void foo(void) {

  rcha = cha & chb;      /* Non-compliant - char type used as a numeric value */
  rcha = cha << 1;       /* Non-compliant - char type used as a numeric value */

  ru8a = u8a & 2U;       /* Compliant */
```

```
  ru8a = u8a << 2U;      /* Compliant */

}
```

In the noncompliant examples, rule 10.1 is violated because the operators & and << do not expect essentially character operands. However, at least one of the operands used here has essentially character type.

Violation of Rule 10.1, Rationale 5: Inappropriate Enum Operands

```
typedef unsigned char boolean;

enum enuma { a1, a2, a3 } rena, ena, enb;

void foo(void) {

  ena--;             /* Non-Compliant - arithmetic operation with enum type*/
  rena = ena * a1;   /* Non-Compliant - arithmetic operation with enum type*/
  ena += a1;         /* Non-Compliant - arithmetic operation with enum type*/

}
```

In the noncompliant examples, rule 10.1 is violated because the arithmetic operators - -, * and += do not expect essentially enum operands. However, at least one of the operands used here has essentially enum type.

Violation of Rule 10.1, Rationale 6: Inappropriate Signed Operand for Bitwise Operations

```
extern signed char s8a;
extern unsigned char ru8a, u8a;

void foo(void) {

  ru8a = s8a & 2;       /* Non-compliant - bitwise operation on signed type */
  ru8a = 2 << 3U;       /* Non-compliant - shift operation on signed type */

  ru8a = u8a << 2U;     /* Compliant */

}
```

In the noncompliant examples, rule 10.1 is violated because the & and << operations must not be performed on essentially signed operands. However, the operands used here are signed.

Violation of Rule 10.1, Rationale 7: Inappropriate Signed Right Operand for Shift Operations

```
extern signed char s8a;
extern unsigned char ru8a, u8a;

void foo(void) {

  ru8a = u8a << s8a;    /* Non-compliant - shift magnitude uses signed type */
  ru8a = u8a << -1;     /* Non-compliant - shift magnitude uses signed type */

  ru8a = u8a << 2U;     /* Compliant */
  ru8a = u8a << 1;      /* Compliant - exception */

}
```

In the noncompliant examples, rule 10.1 is violated because the operation << does not expect an essentially signed right operand. However, the right operands used here are signed.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 10.2` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”
“Essential Types in MISRA C: 2012 Rules 10.x”

MISRA C:2012 Rule 10.2

Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations

Description

Rule Definition

Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.

Rationale

Essentially character type expressions are `char` variables. Do not use `char` in arithmetic operations because the data does not represent numeric values.

It is appropriate to use `char` with addition and subtraction operations only in the following cases:

- When one operand of the addition (+) operation is a `char` and the other is a signed or unsigned `char`, `short`, `int`, `long` or `long long`. In this case, the operation returns a `char`.
- When the first operand of the subtraction (-) operation is a `char` and the second is a signed or unsigned `char`, `short`, `int`, `long` or `long long`. If both operands are `char`, the operation returns a *standard* type. Otherwise, the operation returns a `char`.

The above uses allow manipulation of character data such as conversion between lowercase and uppercase characters or conversion between digits and their ordinal values.

Additional Message in Report

- The *`operand_name`* operand of the + operator applied to an expression of essentially character type shall have essentially signed or unsigned type.
- The right operand of the `-` operator applied to an expression of essentially character type shall have essentially signed or unsigned or character type.
- The left operand of the `-` operator shall have essentially character type if the right operand has essentially character type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Inappropriate use of `char` with Addition and Subtraction Operators

```
extern uint8_t u8a;
extern int8_t s8a;
extern int16_t s16a;
extern int32_t s32a;
extern float32_t fla;

void foo ( void )
```

```
{
    char cha;

    s16a = s16a - 'a';  /* Noncompliant*/

    cha = '0' + fla;    /* Noncompliant*/

    cha = cha  + ':';   /* Noncompliant*/
}
```

- You cannot subtract a `char`-type variable from an integer. When you subtract `'a'` from the integer `s16a`, Polyspace raises a violation.
- In addition operations, `char` type variables can only be added to integer type variables. When you add the floating point number `fla` to `'0'`, Polyspace raises a violation.
- The arithmetic operation `cha+':'` is not a conversion from upper to lower case or from digit to cardinal value. Polyspace raises a violation when `char` variables are used in arithmetic expressions.

Permissible use of `char` in Arithmetic Operation

```
extern uint8_t u8a;
extern int8_t s8a;
extern int16_t s16a;
extern int32_t s32a;
extern float32_t fla;

void foo ( void )
{
    char cha;

    cha = '0' + u8a;     /* Compliant*/

    cha = s8a + '0';     /* Compliant*/

    s32a = cha  - '0';   /* Compliant*/

    cha = '0' - s8a;     /* Compliant*/

    cha++;               /* Compliant*/
}
```

`char` type variables can be used in certain addition or subtraction operations to perform `char` data manipulations. For instance:

- You can add an unsigned integer `u8a` to the `char` type data `'0'` to convert from `'0'` to a different character.
- Similarly, you can add the signed integer `s8a` to `'0'` to perform a desired character conversion.
- You can also subtract `s8a` from the `char` data `'0'`.
- Incrementing and decrementing `char` data is also permissible.

Check Information

Group: The Essential Type Model
Category: Required

AGC Category: Advisory

See Also

`MISRA C:2012 Rule 10.1` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”
“Essential Types in MISRA C: 2012 Rules 10.x”

MISRA C:2012 Rule 10.3

The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category

Description

Rule Definition

The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.

Rationale

The use of implicit conversions between types can lead to unintended results, including possible loss of value, sign, or precision.

Additional Message in Report

- The expression is assigned to an object with a different essential type category.
- The expression is assigned to an object with a narrower essential type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 10.4` | `MISRA C:2012 Rule 10.5` | `MISRA C:2012 Rule 10.6` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”
“Essential Types in MISRA C: 2012 Rules 10.x”

MISRA C:2012 Rule 10.4

Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category

Description

Rule Definition

Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.

Rationale

The use of implicit conversions between types can lead to unintended results, including possible loss of value, sign, or precision.

Polyspace Implementation

Polyspace does not produce a violation of this rule:

- If one of the operands is the constant zero.
- If one of the operands is a signed constant and the other operand is unsigned, and the signed constant has the same representation as its unsigned equivalent.

  For instance, the statement `u8b = u8a + 3;`, where `u8a` and `u8b` are `unsigned char` variables, does not violate the rule because the constants `3` and `3U` have the same representation.

Additional Message in Report

Operands of *`operator_name`* operator shall have the same essential type category.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 10.3|MISRA C:2012 Rule 10.7|Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”
“Essential Types in MISRA C: 2012 Rules 10.x”

MISRA C:2012 Rule 10.5

The value of an expression should not be cast to an inappropriate essential type

Description

Rule Definition

The value of an expression should not be cast to an inappropriate essential type.

Rationale

Converting Between Variable Types

		From					
		Boolean	**character**	**enum**	**signed**	**unsigned**	**floating**
To	**Boolean**		Avoid	Avoid	Avoid	Avoid	Avoid
	character	Avoid					Avoid
	enum	Avoid	Avoid	Avoid	Avoid	Avoid	Avoid
	signed	Avoid					
	unsigned	Avoid					
	floating	Avoid	Avoid				

Some inappropriate explicit casts are:

- In C99, the result of a cast of assignment to `_Bool` is always 0 or 1. This result is not necessarily the case when casting to another type which is defined as essentially Boolean.
- A cast to an essential enum type may result in a value that does not lie within the set of enumeration constants for that type.
- A cast from essential Boolean to any other type is unlikely to be meaningful.
- Converting between floating and character types is not meaningful as there is no precise mapping between the two representations.

Some acceptable explicit casts are:

- To change the type in which a subsequent arithmetic operation is performed.
- To truncate a value deliberately.
- To make a type conversion explicit in the interests of clarity.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: The Essential Type Model
Category: Advisory
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 10.3` | `MISRA C:2012 Rule 10.8` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”
“Essential Types in MISRA C: 2012 Rules 10.x”

MISRA C:2012 Rule 10.6

The value of a composite expression shall not be assigned to an object with wider essential type

Description

Rule Definition

The value of a composite expression shall not be assigned to an object with wider essential type.

Rationale

A *composite expression* is a nonconstant expression using a composite operator. In the Essential Type Model, composite operators are:

- Multiplicative (`*`, `/`, `%`)
- Additive (binary `+`, binary `-`)
- Bitwise (`&`, `|`, `^`)
- Shift (`<<`, `>>`)
- Conditional (`?`, `:`)

If you assign the result of a composite expression to a larger type, the implicit conversion can result in loss of value, sign, precision, or layout.

Additional Message in Report

The composite expression is assigned to an object with a wider essential type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 10.3` | `MISRA C:2012 Rule 10.7` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”
“Essential Types in MISRA C: 2012 Rules 10.x”

MISRA C:2012 Rule 10.7

If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type

Description

Rule Definition

If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed, then the other operand shall not have wider essential type.

Rationale

A *composite expression* is a nonconstant expression using a composite operator. In the Essential Type Model, composite operators are:

- Multiplicative (`*`, `/`, `%`)
- Additive (binary `+`, binary `-`)
- Bitwise (`&`, `|`, `^`)
- Shift (`<<`, `>>`)
- Conditional (`?`, `:`)

Restricting implicit conversion on composite expressions mean that sequences of arithmetic operations within expressions must use the same essential type. This restriction reduces confusion and avoids loss of value, sign, precision, or layout. However, this rule does not imply that all operands in an expression are of the same essential type.

Additional Message in Report

- The right operand shall not have wider essential type than the left operand which is a composite expression.
- The left operand shall not have wider essential type than the right operand which is a composite expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”

“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”
“Essential Types in MISRA C: 2012 Rules 10.x”

MISRA C:2012 Rule 10.8

The value of a composite expression shall not be cast to a different essential type category or a wider essential type

Description

Rule Definition

The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Rationale

A *composite expression* is a non-constant expression using a composite operator. In the Essential Type Model, composite operators are:

- Multiplicative (`*`, `/`, `%`)
- Additive (binary `+`, binary `-`)
- Bitwise (`&`, `|`, `^`)
- Shift (`<<`, `>>`)
- Conditional (`?`, `:`)

Casting to a wider type is not permitted because the result may vary between implementations. Consider this expression:

```
(uint32_t) (u16a +u16b);
```

On a 16-bit machine the addition is performed in 16 bits. The result is wrapped before it is cast to 32 bits. On a 32-bit machine, the addition takes place in 32 bits and preserves high-order bits that are lost on a 16-bit machine. Casting to a narrower type with the same essential type category is acceptable as the explicit truncation of the results always leads to the same loss of information.

For information on essential types, see `MISRA C:2012 Rule 10.1`.

Polyspace Implementation

The rule checker raises a defect only if the result of a composite expression is cast to a different or wider essential type.

For instance, in this example, a violation is shown in the first assignment to `i` but not the second. In the first assignment, a composite expression `i+1` is directly cast from a signed to an unsigned type. In the second assignment, the composite expression is first cast to the same type and then the result is cast to a different type.

```
typedef int int32_T;
typedef unsigned char uint8_T;
...
...
int32_T i;
i = (uint8_T)(i+1); /* Noncompliant */
i = (uint8_T)((int32_T)(i+1)); /* Compliant */
```

Additional Message in Report

- The value of a composite expression shall not be cast to a different essential type category.
- The value of a composite expression shall not be cast to a wider essential type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Casting to Different or Wider Essential Type

```
extern unsigned short ru16a, u16a, u16b;
extern unsigned int   u32a, ru32a;
extern signed int     s32a, s32b;

void foo(void)
{
  ru16a  = (unsigned short) (u32a + u32a);/* Compliant                          */
  ru16a += (unsigned short) s32a;    /* Compliant - s32a is not composite       */
  ru32a  = (unsigned int) (u16a + u16b);  /* Noncompliant - wider essential type */
}
```

In this example, rule 10.8 is violated in the following cases:

- `s32a` and `s32b` are essentially `signed` variables. However, the result `( s32a + s32b )` is cast to an essentially `unsigned` type.
- `u16a` and `u16b` are essentially `unsigned short` variables. However, the result `( s32a + s32b )` is cast to a wider essential type, `unsigned int`.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 10.5` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 11.1

Conversions shall not be performed between a pointer to a function and any other type

Description

Rule Definition

Conversions shall not be performed between a pointer to a function and any other type.

Rationale

The rule forbids the following two conversions:

- Conversion from a function pointer to any other type. This conversion causes undefined behavior.
- Conversion from a function pointer to another function pointer, if the function pointers have different argument and return types.

  The conversion is forbidden because calling a function through a pointer with incompatible type results in undefined behavior.

Polyspace Implementation

Polyspace considers both explicit and implicit casts when checking this rule. However, casts from `NULL` or `(void*)0` do not violate this rule.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Cast between two function pointers

```
typedef void (*fp16) (short n);
typedef void (*fp32) (int n);

#include <stdlib.h>                     /* To obtain macro  NULL */

void func(void) {   /* Exception 1 - Can convert a null pointer
                     * constant into a pointer to a function */
  fp16 fp1 = NULL;                 /* Compliant - exception  */
  fp16 fp2 = (fp16) fp1;           /* Compliant */
  fp32 fp3 = (fp32) fp1;           /* Non-compliant */
  if (fp2 != NULL) {}              /* Compliant - exception  */
  fp16 fp4 = (fp16) 0x8000;        /* Non-compliant - integer to
                                    * function pointer */}
```

In this example, the rule is violated when:

- The pointer `fp1` of type `fp16` is cast to type `fp32`. The function pointer types `fp16` and `fp32` have different argument types.

- An integer is cast to type `fp16`.

The rule is not violated when function pointers `fp1` and `fp2` are cast to `NULL`.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 11.2

Conversions shall not be performed between a pointer to an incomplete type and any other type

Description

Rule Definition

Conversions shall not be performed between a pointer to an incomplete type and any other type.

Rationale

An incomplete type is a type that does not contain sufficient information to determine its size. For example, the statement `struct s;` describes an incomplete type because the fields of `s` are not defined. The size of a variable of type `s` cannot be determined.

Conversions to or from a pointer to an incomplete type result in undefined behavior. Typically, a pointer to an incomplete type is used to hide the full representation of an object. This encapsulation is broken if another pointer is implicitly or explicitly cast to such a pointer.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Casts from incomplete type

```
struct s *sp;
struct t *tp;
short  *ip;
struct ct *ctp1;
struct ct *ctp2;


void foo(void) {

    ip = (short *) sp;            /* Non-compliant */
    sp = (struct s *) 1234;       /* Non-compliant */
    tp = (struct t *) sp;         /* Non-compliant */
    ctp1 = (struct ct *) ctp2;    /* Compliant */

    /* You can convert a null pointer constant to
     * a pointer to an incomplete type */
    sp = NULL;                    /* Compliant - exception  */

    /* A pointer to an incomplete type may be converted into void */
    struct s *f(void);
    (void) f();                   /* Compliant - exception  */

}
```

In this example, types `s`, `t` and `ct` are incomplete. The rule is violated when:

- The variable `sp` with an incomplete type is cast to a basic type.
- The variable `sp` with an incomplete type is cast to a different incomplete type `t`.

The rule is not violated when:

- The variable `ctp2` with an incomplete type is cast to the same incomplete type.
- The `NULL` pointer is cast to the variable `sp` with an incomplete type.
- The return value of `f` with incomplete type is cast to `void`.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 11.5` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 11.3

A cast shall not be performed between a pointer to object type and a pointer to a different object type

Description

Rule Definition

A cast shall not be performed between a pointer to object type and a pointer to a different object type.

Rationale

If a pointer to an object is cast into a pointer to a different object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.

Even if the conversion produces a pointer that is correctly aligned, the behavior can be undefined if the pointer is used to access an object.

Exception: You can convert a pointer to object type into a pointer to one of the following types:

- `char`
- `signed char`
- `unsigned char`

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Noncompliant: Cast to Pointer Pointing to Object of Wider Type

```
signed   char *p1;
unsigned int *p2;

void foo(void){
  p2 = ( unsigned int * ) p1;     /* Non-compliant */
}
```

In this example, `p1` can point to a `signed char` object. However, `p1` is cast to a pointer that points to an object of wider type, `unsigned int`.

Noncompliant: Cast to Pointer Pointing to Object of Narrower Type

```
extern unsigned int read_value ( void );
extern void display ( unsigned int n );

void foo ( void ){
  unsigned int u = read_value ( );
  unsigned short *hi_p = ( unsigned short * ) &u;    /* Non-compliant  */
  *hi_p = 0;
```

```
  display ( u );
}
```

In this example, `u` is an `unsigned int` variable. `&u` is cast to a pointer that points to an object of narrower type, `unsigned short`.

On a big-endian machine, the statement `*hi_p = 0` attempts to clear the high bits of the memory location that `&u` points to. But, from the result of `display(u)`, you might find that the high bits have not been cleared.

Compliant: Cast Adding a Type Qualifier

```
const short *p;
const volatile short *q;
void foo (void){
  q = ( const volatile short * ) p;  /* Compliant */
}
```

In this example, both `p` and `q` can point to `short` objects. The cast between them adds a `volatile` qualifier only and is therefore compliant.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 11.4` | `MISRA C:2012 Rule 11.5` | `MISRA C:2012 Rule 11.8` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 11.4

A conversion should not be performed between a pointer to object and an integer type

Description

Rule Definition

A conversion should not be performed between a pointer to object and an integer type.

Rationale

Conversion between integers and pointers can cause errors or undefined behavior.

- If an integer is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.
- If a pointer is cast to an integer, the resulting value can be outside the allowed range for the integer type.

Polyspace Implementation

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Casts between pointer and integer

```
#include <stdbool.h>

typedef unsigned char      uint8_t;
typedef          char      char_t;
typedef unsigned short     uint16_t;
typedef signed   int       int32_t;

typedef _Bool bool_t;
uint8_t *PORTA = (uint8_t *) 0x0002;            /* Non-compliant */

void foo(void) {

    char_t c = 1;
    char_t *pc = &c;                              /* Compliant */


    uint16_t ui16   = 7U;
    uint16_t *pui16 = &ui16;                      /* Compliant */
    pui16 = (uint16_t *) ui16;                    /* Non-compliant */


    uint16_t *p;
```

```
    int32_t addr = (int32_t) p;                 /* Non-compliant */
    bool_t b = (bool_t) p;                      /* Non-compliant */
    enum etag { A, B } e = ( enum etag ) p;     /* Non-compliant */
}
```

In this example, the rule is violated when:

- The integer `0x0002` is cast to a pointer.

  If the integer defines an absolute address, it is more common to assign the address to a pointer in a header file. To avoid the assignment being flagged, you can then exclude headers files from coding rules checking. For more information, see `Do not generate results for (-do-not-generate-results-for)`.
- The pointer `p` is cast to integer types such as `int32_t`, `bool_t` or `enum etag`.

The rule is not violated when the address `&ui16` is assigned to a pointer.

Check Information

Group: Pointer Type Conversions
Category: Advisory
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 11.3|MISRA C:2012 Rule 11.7|MISRA C:2012 Rule 11.9|Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 11.5

A conversion should not be performed from pointer to void into pointer to object

Description

Rule Definition

A conversion should not be performed from pointer to void into pointer to object.

Rationale

If a pointer to `void` is cast into a pointer to an object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior. However, such a cast can sometimes be necessary, for example, when using memory allocation functions.

Polyspace Implementation

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Cast from Pointer to `void`

```
void foo(void) {

    unsigned int  u32a = 0;
    unsigned int  *p32 = &u32a;
    void          *p;
    unsigned int  *p16;

    p   = p32;                /* Compliant - pointer to uint32_t
                               *             into pointer to void */
    p16 = p;                  /* Non-compliant */

    p   = (void *) p16;       /* Compliant */
    p32 = (unsigned int *) p; /* Non-compliant */
}
```

In this example, the rule is violated when the pointer `p` of type `void*` is cast to pointers to other types.

The rule is not violated when `p16` and `p32`, which are pointers to non-`void` types, are cast to `void*`.

Check Information

Group: Pointer Type Conversions
Category: Advisory
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 11.2` | `MISRA C:2012 Rule 11.3` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 11.6

A cast shall not be performed between pointer to void and an arithmetic type

Description

Rule Definition

A cast shall not be performed between pointer to void and an arithmetic type.

Rationale

Conversion between integer types and pointers to `void` can cause errors or undefined behavior.

- If an integer type is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.
- If a pointer is cast to an arithmetic type, the resulting value can be outside the allowed range for the type.

Conversion between non-integer arithmetic types and pointers to `void` is undefined.

Polyspace Implementation

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Casts Between Pointer to `void` and Arithmetic Types

```
void foo(void) {

    void          *p;
    unsigned int  u;
    unsigned short r;

    p = (void *) 0x1234u;            /* Non-compliant - undefined */
    u = (unsigned int) p;            /* Non-compliant - undefined */

    p = (void *) 0;                  /* Compliant - Exception */

}
```

In this example, `p` is a pointer to `void`. The rule is violated when:

- An integer value is cast to `p`.
- `p` is cast to an `unsigned int` type.

The rule is not violated if an integer constant with value 0 is cast to a pointer to `void`.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 11.7

A cast shall not be performed between pointer to object and a non-integer arithmetic type

Description

Rule Definition

A cast shall not be performed between pointer to object and a non-integer arithmetic type.

Rationale

This rule covers types that are essentially Boolean, character, enum or floating.

- If an essentially Boolean, character or enum variable is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior. If a pointer is cast to one of those types, the resulting value can be outside the allowed range for the type.
- Casts to or from a pointer to a floating type results in undefined behavior.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Casts from Pointer to Non-Integer Arithmetic Types

```
int foo(void) {

    short *p;
    float  f;
    long  *l;

    f = (float)   p;              /* Non-compliant */
    p = (short *) f;              /* Non-compliant */

    l = (long *)  p;              /* Compliant */
}
```

In this example, the rule is violated when:

- The pointer `p` is cast to `float`.
- A `float` variable is cast to a pointer to `short`.

The rule is not violated when the pointer `p` is cast to `long*`.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 11.4` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 11.8

A cast shall not remove any const or volatile qualification from the type pointed to by a pointer

Description

Rule Definition

A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.

Rationale

This rule forbids:

- Casts from a pointer to a `const` object to a pointer that does not point to a `const` object.
- Casts from a pointer to a `volatile` object to a pointer that does not point to a `volatile` object.

Such casts violate type qualification. For example, the `const` qualifier indicates the read-only status of an object. If a cast removes the qualifier, the object is no longer read-only.

Polyspace Implementation

Polyspace flags both implicit and explicit conversions that violate this rule.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Casts That Remove Qualifiers

```
void foo(void) {

    /* Cast on simple type */
    unsigned short           x;
    unsigned short * const   cpi = &x;  /* const pointer */
    unsigned short * const  *pcpi;   /* pointer to const pointer */
    unsigned short **ppi;
    const unsigned short    *pci;    /* pointer to const */
    volatile unsigned short *pvi;    /* pointer to volatile  */
    unsigned short          *pi;

    pi = cpi;                        /* Compliant - no cast required */
    pi  = (unsigned short *)  pci;   /* Non-compliant */
    pi  = (unsigned short *)  pvi;   /* Non-compliant */
    ppi = (unsigned short **)pcpi;   /* Non-compliant */
}
```

In this example:

- The variables `pci` and `pcpi` have the `const` qualifier in their type. The rule is violated when the variables are cast to types that do not have the `const` qualifier.

- The variable `pvi` has a `volatile` qualifier in its type. The rule is violated when the variable is cast to a type that does not have the `volatile` qualifier.

Even though `cpi` has a `const` qualifier in its type, the rule is not violated in the statement `p=cpi;`. The assignment does not cause a type conversion because both `p` and `cpi` have type `unsigned short`.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 11.3` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 11.9

The macro NULL shall be the only permitted form of integer null pointer constant

Description

Rule Definition

The macro NULL shall be the only permitted form of integer null pointer constant.

Rationale

The following expressions allow the use of a null pointer constant:

- Assignment to a pointer
- The == or != operation, where one operand is a pointer
- The ?: operation, where one of the operands on either side of : is a pointer

Using `NULL` rather than 0 makes it clear that a null pointer constant was intended.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Using 0 for Pointer Assignments and Comparisons

```
void main(void) {

    int *p1 = 0;              /* Non-compliant */
    int *p2 = ( void * ) 0;   /* Compliant     */

#define MY_NULL_1 0
#define MY_NULL_2 ( void * ) 0

    if ( p1 == MY_NULL_1 )    /* Non-compliant */
    { }
    if ( p2 == MY_NULL_2 )    /* Compliant     */
    { }

}
```

In this example, the rule is violated when the constant 0 is used instead of `(void*) 0` for pointer assignments and comparisons.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Readability

See Also

`MISRA C:2012 Rule 11.4` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 12.1

The precedence of operators within expressions should be made explicit

Description

Rule Definition

The precedence of operators within expressions should be made explicit.

Rationale

The C language has a large number of operators and their precedence is not intuitive. Inexperienced programmers can easily make mistakes. Remove any ambiguity by using parentheses to explicitly define operator precedence.

The following table list the MISRA C definition of operator precedence for this rule.

Description	Operator and Operand	Precedence
Primary	identifier, constant, string literal, (expression)	16
Postfix	`[]` `()` (function call) `.` `->` ++(post-increment) `--`(post-decrement) `()` `{}`(C99: compound literals)	15
Unary	++(post-increment) `--`(post-decrement) `&` `*` `+` `-` `~` `!` sizeof defined (preprocessor)	14
Cast	`()`	13
Multiplicative	`*` `/` `%`	12
Additive	`+` `-`	11
Bitwise shift	`<<` `>>`	10
Relational	`<>` `<=` `>=`	9
Equality	`==` `!=`	8
Bitwise AND	&	7
Bitwise XOR	^	6
Bitwise OR	|	5
Logical AND	&&	4
Logical OR	||	3
Conditional	`?:`	2
Assignment	`=` `*=` `/=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`	1
Comma	,	0

Additional Message in Report

Operand of logical %s is not a primary expression. The precedence of operators within expressions should be made explicit.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Ambiguous Precedence in Multi-Operation Expressions

```
int a, b, c, d, x;

void foo(void) {
  x = sizeof a + b;                     /* Non-compliant - MISRA-12.1 */

  x = a == b ? a : a - b;               /* Non-compliant - MISRA-12.1 */

  x = a <<  b + c ;                     /* Non-compliant - MISRA-12.1 */

  if (a || b && c) { }                  /* Non-compliant - MISRA-12.1 */

  if ( (a>x) && (b>x) || (c>x) )   { }  /* Non-compliant - MISRA-12.1 */
}
```

This example shows various violations of MISRA rule 12.1. In each violation, if you do not know the order of operations, the code could execute unexpectedly.

Correction — Clarify With Parentheses

To comply with this MISRA rule, add parentheses around individual operations in the expressions. One possible solution is shown here.

```
int a, b, c, d, x;

void foo(void) {
  x = sizeof(a) + b;

  x = ( a == b ) ? a : ( a - b );

  x = a << ( b + c );

  if ( ( a || b ) && c) { }

  if ( ((a>x) && (b>x)) || (c>x) ) { }
}
```

Ambiguous Precedence In Preprocessing Expressions

```
# if defined X && X + Y > Z    /* Non-compliant - MISRA-12.1 */
# endif

# if ! defined X && defined Y  /* Non-compliant - MISRA-12.1 */
# endif
```

In this example, two violations of MISRA rule 12.1 are shown in preprocessing code. In each violation, if you do not know the correct order of operations, the results can be unexpected and cause problems.

Correction — Clarify with Parentheses

To comply with this MISRA rule, add parentheses around individual operations in the expressions. One possible solution is shown here.

```
# if defined (X) && ( (X + Y) > Z )
# endif

# if ! defined (X) && defined (Y)
# endif
```

Compliant Expressions Without Parentheses

```
int a, b, c, x;
struct {int a; } s, *ps, *pp[2];

void foo(void) {
  ps = &s

  pp[i]-> a;          /* Compliant - no need to write (pp[i])->a */
  *ps++;              /* Compliant - no need to write *( p++ ) */

  x = f ( a + b, c ); /* Compliant - no need to write f ( (a+b),c) */

  x = a, b;           /* Compliant - parsed as ( x = a ), b */

  if (a && b && c ){  /* Compliant - all operators have
                       * the same precedence */
}
```

In this example, the expressions shown have multiple operations. However, these expressions are compliant because operator precedence is already clear.

Check Information

Group: Expressions
Category: Advisory
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 12.2` | `MISRA C:2012 Rule 12.3` | `MISRA C:2012 Rule 12.4` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 12.2

The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand

Description

Rule Definition

The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand.

Rationale

Consider the following statement:

```
var = abc << num;
```

If `abc` is a 16-bit integer, then `num` must be in the range 0–15, (nonnegative and less than 16). If `num` is negative or greater than 16, then the shift behavior is undefined.

Polyspace Implementation

In Polyspace, the numbers that are manipulated in preprocessing directives are 64 bits wide. The valid shift range is between 0 and 63. When bitfields are within a complex expression, Polyspace extends this check onto the bitfield field width or the width of the base type.

Additional Message in Report

- Shift amount is bigger than *`size`*.
- Shift amount is negative.
- The right operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left operand.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 12.1` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 12.3

The comma operator should not be used

Description

Rule Definition

The comma operator should not be used.

Rationale

The comma operator can be detrimental to readability. You can often write the same code in another form.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Comma Usage in C Code

```
typedef signed int abc, xyz, jkl;

static void func1 ( abc, xyz, jkl );       /* Compliant - case 1 */

int foo(void)
{
    volatile int rd = 1;                   /* Compliant - case 2*/
    int var=0, foo=0, k=0, n=2, p, t[10];  /* Compliant - case 3*/

    int abc = 0, xyz = abc + 1;            /* Compliant - case 4*/
    int jkl = ( abc + xyz, abc + xyz );    /* Not compliant - case 1*/

    var = 1, foo += var, kkk = 3;          /* Not compliant - case 2*/
    var = (kkk = 1, foo = 2);              /* Not compliant - case 3*/

    for ( var = 0, ptr = &t[ 0 ]; var < num; ++var, ++ptr){}
                                           /* Not compliant - case 4*/

    if ((abc,xyz)<0) { return 1; }         /* Not compliant - case 5*/
}
```

In this example, the code shows various uses of commas in C code.

Noncompliant Cases

Case	Reason for noncompliance
1	When reading the code, it is not immediately obvious what `jkl` is initialized to. For example, you could infer that `jkl` has a value `abc+xyz`, `(abc+xyz)*(abc+xyz)`, `f((abc+xyz),(abc+xyz))`, and so on.

Case	Reason for noncompliance
2	When reading the code, it is not immediately obvious whether `foo` has a value 0 or 1 after the statement.
3	When reading the code, it is not immediately obvious what value is assigned to `var`.
4	When reading the code, it is not immediately obvious which values control the `for` loop.
5	When reading the code, it is not immediately obvious whether the `if` statement depends on `abc`, `xyz`, or both.

Compliant Cases

Case	Reason for compliance
1	Using commas to call functions with variables is allowed.
2	Comma operator is not used.
3 & 4	When using the comma for initialization, the variables and their values are immediately obvious.

Check Information

Group: Expressions
Category: Advisory
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 12.1` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 12.4

Evaluation of constant expressions should not lead to unsigned integer wrap-around

Description

Rule Definition

Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Rationale

Unsigned integer expressions do not strictly overflow, but instead wraparound. Although there may be good reasons to use modulo arithmetic at run time, intentional use at compile time is less likely.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Advisory
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 12.1` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 12.5

The `sizeof` operator shall not have an operand which is a function parameter declared as “array of type”

Description

Rule Definition

The `sizeof` operator shall not have an operand which is a function parameter declared as “array of type”.

Rationale

The `sizeof` operator acting on an array normally returns the array size in bytes. For instance, in the following code, `sizeof(arr)` returns the size of `arr` in bytes.

```
int32_t arr[4];
size_t numberOfElements = sizeof (arr) / sizeof(arr[0]);
```

However, when the array is a function parameter, it degenerates to a pointer. The `sizeof` operator acting on the array returns the corresponding pointer size and not the array size.

The use of `sizeof` operator on an array that is a function parameter typically indicates an unintended programming error.

Additional Message in Report

The `sizeof` operator shall not have an operand which is a function parameter declared as “array of type”.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Incorrect Use of `sizeof` Operator

```
int32_t glbA[] = { 1, 2, 3, 4, 5 };
void f (int32_t A[4])
{
         uint32_t numElements = sizeof(A) / sizeof(int32_t);  /* Non-compliant */
        uint32_t numElements_glbA = sizeof(glbA) / sizeof(glbA[0]);  /* Compliant */
}
```

In this example, the variable `numElements` always has the same value of 1, irrespective of the number of members that appear to be in the array (4 in this case), because `A` has type `int32_t *` and not `int32_t[4]`.

The variable `numElements_glbA` has the expected vale of 5 because the `sizeof` operator acts on the global array `glbA`.

Check Information

Group: Expressions
Category: Mandatory
AGC Category: Mandatory

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Rule 13.1

Initializer lists shall not contain persistent side effects

Description

Rule Definition

Initializer lists shall not contain persistent side effects.

Rationale

C99 permits initializer lists with expressions that can be evaluated only at run-time. However, the order in which elements of the list are evaluated is not defined. If one element of the list modifies the value of a variable which is used in another element, the ambiguity in order of evaluation causes undefined values. Therefore, this rule requires that expressions occurring in an initializer list cannot modify the variables used in them.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Initializers with Persistent Side Effect

```
volatile int v;
int x;
int y;

void f(void) {
    int arr[2] = {x+y,x-y};  /* Compliant */
    int arr2[2] = {v,0};      /* Non-compliant */
    int arr3[2] = {x++,y};    /* Non-compliant */
}
```

In this example, the rule is not violated in the first initialization because the initializer does not modify either `x` or `y`. The rule is violated in the other initializations.

- In the second initialization, because `v` is volatile, the initializer can modify `v`.
- In the third initialization, the initializer modifies the variable `x`.

Check Information

Group: Side Effects
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 13.2` | `Check MISRA C:2012 (-misra3)`

Topics
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 13.2

The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders

Description

Rule Definition

The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.

Rationale

If an expression results in different values depending on the order of evaluation, its value becomes implementation-defined.

Polyspace Implementation

An expression can have different values under the following conditions:

- The same variable is modified more than once in the expression, or is both read and written.
- The expression allows more than one order of evaluation.

Therefore, this rule forbids expressions where a variable is modified more than once and can cause different results under different orders of evaluation.

Additional Message in Report

The value of 'XX' depends on the order of evaluation. The value of volatile 'XX' depends on the order of evaluation because of multiple accesses.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Variable Modified More Than Once in Expression

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);         /* Compliant */
    COPY_ELEMENT (i++);       /* Noncompliant  */
}
```

In this example, the rule is violated by the statement `COPY_ELEMENT(i++)` because `i++` occurs twice and the order of evaluation of the two expressions is unspecified.

Variable Modified and Used in Multiple Function Arguments

```
void f (unsigned int param1, unsigned int param2) {}

void main () {
    unsigned int i=0;
    f ( i++, i );                /* Non-compliant */
}
```

In this example, the rule is violated because it is unspecified whether the operation `i++` occurs before or after the second argument is passed to `f`. The call `f(i++,i)` can translate to either `f(0,0)` or `f(0,1)`.

Multiple Volatile Variables in Expression

```
volatile float res;
volatile float x;
volatile float y;

float xCopy;
float yCopy;


void function4(void) {
    res = x + y;          //Noncompliant
    xCopy = x;
    yCopy = y;
    res = xCopy + yCopy;  //Compliant
}
```

In this example, the expression `x + y` is noncompliant because the expression involves multiple volatile variables. The expression effectively consists of three operations, accessing the value of `x`, accessing the value of `y`, and finally the addition. The values of the volatile variables `x` and `y` can vary depending on which variable is read first. The standard does not specify the order in which the variables are read. Therefore, the result of the expression can be different under the allowed evaluation orders. For instance, it is possible that reading `x` first results in a change in the value of `y`, which is subsequently read.

To avoid the violation, assign the volatile variables to nonvolatile temporary variables and use these temporary variables in the expression.

Check Information

Group: Side Effects
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Dir 4.9|MISRA C:2012 Rule 13.1|MISRA C:2012 Rule 13.3|MISRA C:2012 Rule 13.4|Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 13.3

A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator

Description

Rule Definition

A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.

Rationale

The rule is violated if the following happens in the same line of code:

- The increment or decrement operator acts on a variable.
- Another read or write operation is performed on the variable.

For example, the line `y=x++` violates this rule. The `++` and `=` operator both act on `x`.

Although the operator precedence rules determine the order of evaluation, placing the `++` and another operator in the same line can reduce the readability of the code.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Increment Operator Used in Expression with Other Side Effects

```
int input(void);
int choice(void);
int operation(int, int);

int func() {
    int x = input(), y = input(), res;
    int ch = choice();
    if (choice == -1)
        return(x++);
    if (choice == 0) {
        res = x++ + y++;
        return(res);           /* Non-compliant */
    }
    else if (choice == 1) {
        x++;                   /* Compliant */
        y++;                   /* Compliant */
        return (x+y);
    }
    else {
        res = operation(x++,y);
        return(res);           /* Non-compliant */
```

```
    }
}
```

In this example, the rule is violated when the expressions containing the ++ operator have side effects other than that caused by the operator. For example, in the expression `return(x++)`, the other side-effect is the `return` operation.

Check Information

Group: Side Effects
Category: Advisory
AGC Category: Readability

See Also

`MISRA C:2012 Rule 13.2` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 13.4

The result of an assignment operator should not be used

Description

Rule Definition

The result of an assignment operator should not be used.

Rationale

The rule is violated if the following happens in the same line of code:

- The assignment operator acts on a variable.
- Another read or operation is performed on the result of the assignment.

For example, the line `a[x]=a[x=y];` violates this rule. The `[]` operator acts on the result of the assignment `x=y`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Result of Assignment Used

```
int x, y, b, c, d;
int a[10];
unsigned int bool_var, false=0, true=1;

int foo(void) {

    x = y;            /* Compliant - x is not used */

    a[x] = a[x = y];  /* Non-compliant - Value of x=y is used */

    if ( bool_var = false ) {}
                    /* Non-compliant - bool_var=false is used */

    if ( bool_var == false ) {}   /* Compliant */

    if ( ( 0u == 0u ) || ( bool_var = true ) ) {}
    /* Non-compliant - even though (bool_var=true) is not evaluated */

    if ( ( x = f () ) != 0 ) {}
                  /* Non-compliant - value of x=f() is used */

    a[b += c] = a[b];
                  /* Non-compliant - value of b += c is used */

    b = c = d = 0; /* Non-compliant - value of d=0 and c=d=0 are used */
```

```
}
```

In this example, the rule is violated when the result of an assignment is used.

Check Information

Group: Side Effects
Category: Advisory
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 13.2` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 13.5

The right hand operand of a logical && or || operator shall not contain persistent side effects

Description

Rule Definition

The right hand operand of a logical && or || operator shall not contain persistent side effects.

Rationale

The right operand of an || operator is not evaluated if the left operand is true. The right operand of an && operator is not evaluated if the left operand is false. In these cases, if the right operand modifies the value of a variable, the modification does not take place. Following the operation, if you expect a modified value of the variable, the modification might not always happen.

Polyspace Implementation

- For this rule, Polyspace considers that a function call does not have a persistent side effect if the function body is not present in the same file as the function call.

  If a call to a pure function is flagged, before ignoring this rule violation, make sure that the function has no side effects. For instance, floating-point functions such as `abs()` seem to only return a value and have no other side effect. However, these functions make use of the FPU Register Stack and can have side-effects in certain architectures, for instance, certain Intel® architectures.
- If the right operand is a volatile variable, Polyspace does not flag this as a rule violation.

Additional Message in Report

The right hand operand of a && operator shall not contain side effects. The right hand operand of a || operator shall not contain side effects.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Right Operand of Logical Operator with Persistent Side Effects

```
int check (int arg) {
    static int count;
    if(arg > 0) {
        count++;                    /* Persistent side effect */
        return 1;
    }
    else
        return 0;
}
```

```
int getSwitch(void);
int getVal(void);

void main(void) {
    int val = getVal();
    int mySwitch = getSwitch();
    int checkResult;

    if(mySwitch && check(val)) {   /* Non-compliant */
    }

    checkResult = check(val);
    if(checkResult && mySwitch) {  /* Compliant */
    }

    if(check(val) && mySwitch) {   /* Compliant */
    }
}
```

In this example, the rule is violated when the right operand of the && operation contains a function call. The function call has a persistent side effect because the static variable `count` is modified in the function body. Depending on `mySwitch`, this modification might or might not happen.

The rule is not violated when the left operand contains a function call. Alternatively, to avoid the rule violation, assign the result of the function call to a variable. Use this variable in the logical operation in place of the function call.

In this example, the function call has the side effect of modifying a `static` variable. Polyspace flags all function calls when used on the right-hand side of a logical && or || operator, even when the function does not have a side effect. Manually inspect your function body to see if it has side effects. If the function does not have side effects, add a comment and justification in your Polyspace result explaining why you retained your code.

Check Information

Group: Side Effects
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 13.6

The operand of the sizeof operator shall not contain any expression which has potential side effects

Description

Rule Definition

The operand of the sizeof operator shall not contain any expression which has potential side effects.

Rationale

The argument of a `sizeof` operator is usually not evaluated at run time. If the argument is an expression, you might wrongly expect that the expression is evaluated.

Polyspace Implementation

The rule is not violated if the argument is a `volatile` variable.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Expressions in `sizeof` Operator

```
#include <stddef.h>
int x;
int y[40];
struct S {
    int a;
    int b;
};
struct S myStruct;

void main() {
    size_t sizeOfType;
    sizeOfType = sizeof(x);         /* Compliant */
    sizeOfType = sizeof(y);         /* Compliant */
    sizeOfType = sizeof(myStruct);  /* Compliant */
    sizeOfType = sizeof(x++);       /* Non-compliant */
}
```

In this example, the rule is violated when the expression `x++` is used as argument of `sizeof` operator.

Check Information

Group: Side Effects
Category: Mandatory
AGC Category: Mandatory

See Also

`MISRA C:2012 Rule 18.8` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 14.1

A loop counter shall not have essentially floating type

Description

Rule Definition

A loop counter shall not have essentially floating type.

Rationale

When using a floating-point loop counter, accumulation of rounding errors can result in a mismatch between the expected and actual number of iterations. This rounding error can happen when a loop step that is not a power of the floating point radix is rounded to a value that can be represented by a float.

Even if a loop with a floating-point loop counter appears to behave correctly on one implementation, it can give a different number of iteration on another implementation.

Polyspace Implementation

If the `for` index is a variable symbol, Polyspace checks that it is not a float.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

`for` Loop Counters

```
int main(void){
    unsigned int counter = 0u;
    int result = 0;
    float foo;

    // Float loop counters
    for(float foo = 0.0f; foo < 1.0f; foo +=0.001f){
        /* Non-compliant - counter = 1000 at the end of the loop */
        ++counter;
    }

    float fff = 0.0f;
    for(fff = 0.0f; fff <12.0f; fff += 1.0f){    /* Non-compliant*/
        result++;
    }

    // Integer loop count
    for(unsigned int count = 0u; count < 1000u; ++count){ /* Compliant */
        foo = (float) count * 0.001f;
    }
}
```

In this example, the three `for` loops show three different loop counters. The first and second `for` loops use float variables as loop counters, and therefore are not compliant. The third loop uses the integer `count` as the loop counter. Even though `count` is used as a float inside the loop, the variable remains an integer when acting as the loop index. Therefore, this `for` loop is compliant.

`while` Loop Counters

```
int main(void){
    unsigned int u32a;
    float foo;

    foo = 0.0f;
    while (foo < 1.0f){
        foo += 0.001f;  /* Non-compliant - foo used as a loop counter */
    }

    foo = read_float32();
    do{
        u32a = read_u32();
    }while( ((float)u32a - foo) > 10.0f );
                        /* Compliant - foo doesn't change in the loop */
                        /*  so cannot be a counter */
    return 1;
}
```

This example shows two `while` loops both of which use `foo` in the `while`-loop conditions.

The first `while` loop uses `foo` in the condition and inside the loop. Because `foo` changes, floating-point rounding errors can cause unexpected behavior.

The second `while` loop does not use `foo` inside the loop, but does use `foo` inside the `while`-condition. So `foo` is not the loop counter. The integer `u32a` is the loop counter because it changes inside the loop and is part of the while condition. Because `u32a` is an integer, the rounding error issue is not a concern, making this `while` loop compliant.

Check Information

Group: Control Statement Expressions
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 14.2` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 14.2

A for loop shall be well-formed

Description

Rule Definition

A for loop shall be well-formed.

Rationale

The `for` statement provides a general-purpose looping facility. Using a restricted form of loop makes code easier to review and to analyze.

Polyspace Implementation

Polyspace checks that:

- The `for` loop index (`V`) is a variable symbol.
- `V` is the last assigned variable in the first expression (if present).
- If the first expression exists, it contains an assignment of `V`.
- If the second expression exists, it is a comparison of `V`.
- If the third expression exists, it is an assignment of `V`.
- There are no direct assignments of the `for` loop index.

Additional Message in Report

- 1st expression should be an assignment. The following kinds of for loops are allowed:
  - all three expressions shall be present;
  - the 2nd and 3rd expressions shall be present with prior initialization of the loop counter;
  - all three expressions shall be empty for a deliberate infinite loop.
- 3rd expression should be an assignment of a loop counter.
- 3rd expression : assigned variable should be the loop counter (*`counter`*).
- 3rd expression should be an assignment of loop counter (*`counter`*) only.
- 2nd expression should contain a comparison with loop counter (*`counter`*).
- Loop counter (*`counter`*) should not be modified in the body of the loop.
- Bad type for loop counter (*`counter`*).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Altering the Loop Counter Inside the Loop

```
void foo(void){

    for(short index=0; index < 5; index++){  /* Non-compliant */
        index = index + 3;       /* Altering the loop counter */
    }
}
```

In this example, the loop counter `index` changes inside the `for` loop. It is hard to determine when the loop terminates.

Correction — Use Another Variable to Terminate Early

One possible correction is to use an extra flag to terminate the loop early.

In this correction, the second clause of the `for` loop depends on the counter value, `index < 5`, and upon an additional flag, `!flag`. With the additional flag, the for loop definition and counter remain readable, and you can escape the loop early.

```
#define FALSE 0
#define TRUE  1

void foo(void){

    int flag = FALSE;

    for(short index=0; (index < 5) && !flag; index++){ /* Compliant */
        if((index % 4) == 0){
            flag = TRUE;        /* allows early termination of loop */
        }
    }
}
```

`for` Loops With Empty Clauses

```
void foo(void)
    for(short index = 0; ; index++) {}   /* Non-compliant */

    for(short index = 0; index < 10;) {} /* Non-compliant */

    short index;
    for(; index < 10;) {}     /* Non-compliant */

    for(; index < 10; i++) {} /* Compliant */

    for(;;){}
          /* Compliant - Exception all three clauses can be empty */
}
```

This example shows `for` loops definitions with a variety of missing clauses. To be compliant, initialize the first clause variable before the `for` loop (line 9). However, you cannot have a `for` loop without the second or third clause.

The one exception is a `for` loop with all three clauses empty, so as to allow for infinite loops.

Check Information

Group: Control Statement Expressions
Category: Required
AGC Category: Readability

See Also

`MISRA C:2012 Rule 14.1` | `MISRA C:2012 Rule 14.3` | `MISRA C:2012 Rule 14.4` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 14.3

Controlling expressions shall not be invariant

Description

Rule Definition

Controlling expressions shall not be invariant.

Rationale

If the controlling expression, for example an `if` condition, has a constant value, the non-changing value can point to a programming error.

Polyspace Implementation

The checker flags conditions in `if` or `while` statements or conditions that appear as the first operands of ternary operators (`?:`) if the conditions are invariant, for instance, evaluate always to true or false.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Polyspace Bug Finder flags some violations of MISRA C 14.3 through the `Dead code` and `Useless if` checkers.

Polyspace Code Prover does not use gray code to flag MISRA C 14.3 violations. In Code Prover, you can also see a difference in results based on your choice for the option `Verification level (-to)`. See “Check for Coding Standard Violations”.

Additional Message in Report

- Boolean operations whose results are invariant shall not be permitted.
- Expression is always true.
- Expression is always false.
- Controlling expressions shall not be invariant.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Control Statement Expressions
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 2.1` | `MISRA C:2012 Rule 14.2` | `Check MISRA C:2012 (-misra3)`

Topics
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 14.4

The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type

Description

Rule Definition

The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type

Rationale

Strong typing requires the controlling expression on an `if` statement or iteration statement to have *essentially Boolean* type.

Polyspace Implementation

Polyspace does not flag integer constants, for example `if(2)`.

The analysis recognizes the Boolean types, `bool` or `_Bool` (defined in `stdbool.h`)

You can also define types that are essentially Boolean using the option `Effective boolean types (-boolean-types)`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Controlling Expression in `if`, `while`, and `for`

```
#include <stdbool.h>
#include <stdlib.h>

#define TRUE = 1

typedef _Bool bool_t;
extern bool_t flag;

void foo(void){
    int *p = 1;
    int *q = 0;
    int i = 0;
    while(p){}           /* Non-compliant - p is a pointer */

    while(q != NULL){}   /* Compliant */

    while(TRUE){}        /* Compliant */

    while(flag){}        /* Compliant */
```

```
    if(i){}              /* Non-compliant - int32_t is not boolean */

    if(i != 0){}         /* Compliant */

    for(int i=-10; i;i++){}   /* Non-compliant - int32_t is not boolean */

    for(int i=0; i<10;i++){}  /* Compliant */
}
```

This example shows various controlling expressions in `while`, `if`, and `for` statements.

The noncompliant statements (the first `while`, `if`, and `for` examples), use a single non-Boolean variable. If you use a single variable as the controlling statement, it must be essentially Boolean (lines 17 and 19). Boolean expressions are also compliant with MISRA.

Check Information

Group: Control Statement Expressions
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 14.2` | `MISRA C:2012 Rule 20.8` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 15.1

The goto statement should not be used

Description

Rule Definition

The goto statement should not be used.

Rationale

Unrestricted use of `goto` statements makes the program unstructured and difficult to understand.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `goto` Statements

```
void foo(void) {
    int i = 0, result = 0;

label1:
    for ( i; i < 5; i++ ) {
        if (i > 2) goto label2;     /* Non-compliant */
    }

label2: {
        result++;
        goto label1;                /* Non-compliant */
    }
}
```

In this example, the rule is violated when `goto` statements are used.

Check Information

Group: Control Flow
Category: Advisory
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 15.2` | `MISRA C:2012 Rule 15.3` | `MISRA C:2012 Rule 15.4` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 15.2

The goto statement shall jump to a label declared later in the same function

Description

Rule Definition

The goto statement shall jump to a label declared later in the same function.

Rationale

Unrestricted use of `goto` statements makes the program unstructured and difficult to understand. You can use a forward `goto` statement together with a backward one to implement iterations. Restricting backward `goto` statements ensures that you use only iteration statements provided by the language such as `for` or `while` to implement iterations. This restriction reduces visual complexity of the code.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of Backward `goto` Statements

```
void foo(void) {
    int i = 0, result = 0;

label1:
    for ( i; i < 5; i++ ) {
        if (i > 2) goto label2;   /* Compliant */
    }

label2: {
        result++;
        goto label1;               /* Non-compliant */
    }
}
```

In this example, the rule is violated when a `goto` statement causes a backward jump to `label1`.

The rule is not violated when a `goto` statement causes a forward jump to `label2`.

Check Information

Group: Control Flow
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 15.1` | `MISRA C:2012 Rule 15.3` | `MISRA C:2012 Rule 15.4` | `Check MISRA C:2012 (-misra3)`

Topics
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 15.3

Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement

Description

Rule Definition

Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.

Rationale

Unrestricted use of `goto` statements makes the program unstructured and difficult to understand. Restricting use of `goto` statements to jump between blocks or into nested blocks reduces visual code complexity.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

`goto` Statements Jump Inside Block

```
void f1(int a) {
    if(a <= 0) {
        goto L2;        /* Non-compliant - L2 in different block*/
    }

    goto L1;            /* Compliant - L1 in same block*/

    if(a == 0) {
        goto L1;        /* Compliant - L1 in outer block*/
    }

    goto L2;            /* Non-compliant - L2 in inner block*/

    L1: if(a > 0) {
            L2:;
    }
}
```

In this example, `goto` statements cause jumps to different labels. The rule is violated when:

- The label occurs in a block different from the block containing the `goto` statement.

  The block containing the label neither encloses nor is enclosed by the current block.
- The label occurs in a block enclosed by the block containing the `goto` statement.

The rule is not violated when:

- The label occurs in the same block as the block containing the `goto` statement..
- The label occurs in a block that encloses the block containing the `goto` statement..

`goto` Statements in `switch` Block

```
void f2 ( int x, int z ) {
    int y = 0;

    switch(x) {
    case 0:
        if(x == y) {
            goto L1;  /* Non-compliant - switch-clauses are treated as blocks */
        }
        break;
    case 1:
        y = x;
        L1: ++x;
        break;
    default:
        break;
    }

}
```

In this example, the label for the `goto` statement appears to occur in a block that encloses the block containing the `goto` statement. However, for the purposes of this rule, the software considers that each `case` statement begins a new block. Therefore, the `goto` statement violates the rule.

Check Information

Group: Control Flow
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 15.1|MISRA C:2012 Rule 15.2|MISRA C:2012 Rule 15.4|MISRA C:2012 Rule 16.1|Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 15.4

There should be no more than one break or goto statement used to terminate any iteration statement

Description

Rule Definition

There should be no more than one break or goto statement used to terminate any iteration statement.

Rationale

If you use one `break` or `goto` statement in your loop, you have one secondary exit point from the loop. Restricting number of exits from a loop in this way reduces visual complexity of your code.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

`break` Statements in Inner and Outer Loops

```
volatile int stop;

int func(int *arr, int size, int sat) {
    int i,j;
    int sum = 0;
    for (i=0; i< size; i++) {   /* Compliant  */
        if(sum >= sat)
            break;
        for (j=0; j< i; j++) {  /* Compliant */
            if(stop)
                break;
            sum += arr[j];
        }
    }
}
```

In this example, the rule is not violated in both the inner and outer loop because both loops have one `break` statement each.

`break` and `goto` Statements in Loop

```
volatile int stop;

void displayStopMessage();

int func(int *arr, int size, int sat) {
    int i;
    int sum = 0;
    for (i=0; i< size; i++) {   /* Non-compliant  */
        if(sum >= sat)
            break;
```

```
        if(stop)
            goto L1;
        sum += arr[i];
    }

    L1: displayStopMessage();
}
```

In this example, the rule is violated because the `for` loop has one `break` statement and one `goto` statement.

`goto` Statement in Inner Loop and `break` Statement in Outer Loop

```
volatile int stop;

void displayMessage();

int func(int *arr, int size, int sat) {
    int i,j;
    int sum = 0;
    for (i=0; i< size; i++) {  /* Non-compliant */
        if(sum >= sat)
            break;
        for (j=0; j< i; j++) { /* Compliant */
            if(stop)
                goto L1;
            sum += arr[i];
        }
    }

    L1: displayMessage();
}
```

In this example, the rule is not violated in the inner loop because you can exit the loop only through the one `goto` statement. However, the rule is violated in the outer loop because you can exit the loop through either the `break` statement or the `goto` statement in the inner loop.

Check Information

Group: Control Flow
Category: Advisory
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 15.1` | `MISRA C:2012 Rule 15.2` | `MISRA C:2012 Rule 15.3` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 15.5

A function should have a single point of exit at the end

Description

Rule Definition

A function should have a single point of exit at the end.

Rationale

This rule requires that a `return` statement must occur as the last statement in the function body. Otherwise, the following issues can occur:

- Code following a `return` statement can be unintentionally omitted.
- If a function that modifies some of its arguments has early `return` statements, when reading the code, it is not immediately clear which modifications actually occur.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

More Than One `return` Statement in Function

```
#define MAX ((unsigned int)2147483647)
#define NULL (void*)0

typedef unsigned int bool_t;
bool_t false = 0;
bool_t true = 1;

bool_t f1(unsigned short n, char *p) {          /* Non-compliant */
    if(n > MAX) {
        return false;
    }

    if(p == NULL) {
        return false;
    }

    return true;
}
```

In this example, the rule is violated because there are three `return` statements.

Correction — Use Variable to Store Return Value

One possible correction is to store the return value in a variable and return this variable just before the function ends.

```
#define MAX ((unsigned int)2147483647)
#define NULL (void*)0
```

```
typedef unsigned int bool_t;
bool_t false = 0;
bool_t true = 1;
bool_t return_value;

bool_t f2 (unsigned short n, char *p) {         /* Compliant */
    return_value = true;
    if(n > MAX) {
        return_value = false;
    }

    if(p == NULL) {
        return_value = false;
    }

    return return_value;
}
```

Check Information

Group: Control Flow
Category: Advisory
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 17.4` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 15.6

The body of an iteration-statement or a selection-statement shall be a compound statement

Description

Rule Definition

The body of an iteration-statement or a selection-statement shall be a compound- statement.

Rationale

If the block of code associated with an iteration or selection statement is not contained in braces, you can make mistakes about the association. For example:

- You can wrongly associate a line of code with an iteration or selection statement because of its indentation.
- You can accidentally place a semicolon following the iteration or selection statement. Because of the semicolon, the line following the statement is no longer associated with the statement even though you intended otherwise.

This checker enforces the practice of adding braces following a selection or iteration statement even for a single line in the body. Later, when more lines are added, the developer adding them does not need to note the absence of braces and include them.

Polyspace Implementation

The checker flags `for` loops where the first token following a `for` statement is not a left brace, for instance:

```
for (i=init_val; i > 0; i--)
   if (arr[i] < 0)
      arr[i] = 0;
```

Similar checks are performed for `if`, `else if`, `else`, `switch`, `for` and `do..while` statements.

The second line of the message on the **Result Details** pane indicates which statement is violating the rule. For instance, in the preceding example, there are two violations. The second line of the message points to the `for` loop for one violation and the `if` condition for another.

Additional Message in Report

- The else keyword shall be followed by either a compound statement, or another if statement.
- An if (expression) construct shall be followed by a compound statement.
- The statement forming the body of a while statement shall be a compound statement.
- The statement forming the body of a do ... while statement shall be a compound statement.
- The statement forming the body of a for statement shall be a compound statement.
- The statement forming the body of a switch statement shall be a compound statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Iteration Block

```
int data_available = 1;
void f1(void) {
    while(data_available)                /* Non-compliant */
        process_data();

    while(data_available) {              /* Compliant */
        process_data();
    }
}
```

In this example, the second `while` block is enclosed in braces and does not violate the rule.

Nested Selection Statements

```
void f1(void) {
    if(flag_1)                           /* Non-compliant */
        if(flag_2)                       /* Non-compliant */
            action_1();
    else                                 /* Non-compliant */
            action_2();
}
```

In this example, the rule is violated because the `if` or `else` blocks are not enclosed in braces. Unless indented as above, it is easy to associate the `else` statement with the inner `if`.

Correction — Place Selection Statement Block in Braces

One possible correction is to enclose each block associated with an `if` or `else` statement in braces.

```
void f1(void) {
    if(flag_1) {                         /* Compliant */
        if(flag_2) {                       /* Compliant */
            action_1();
        }
    }
    else {                               /* Compliant */
        action_2();
    }
}
```

Spurious Semicolon After Iteration Statement

```
void f1(void) {
    while(flag_1);                       /* Non-compliant */
    {
        flag_1 = action_1();
    }
}
```

In this example, the rule is violated even though the `while` statement is followed by a block in braces. The semicolon following the `while` statement causes the block to dissociated from the `while` statement.

The rule helps detect such spurious semicolons.

Check Information

Group: Control Flow
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 15.7

All if ... else if constructs shall be terminated with an else statement

Description

Rule Definition

All if ... else if constructs shall be terminated with an else statement.

Rationale

Unless there is a terminating `else` statement in an `if...elseif...else` construct, during code review, it is difficult to tell if you considered all possible results for the `if` condition.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Missing `else` Block

```
int get_flag_1(void);
int get_flag_2(void);
void action_1(void);
void action_2(void);

void f1(void) {
    int flag_1 = get_flag_1(), flag_2 = get_flag_2();
    if(flag_1) {
        action_1();
    }
    else if(flag_2) {
        /* Non-compliant */
        action_2();
    }
}
```

In this example, the rule is violated because the `if ... else if` construct does not have a terminating `else` block.

Correction — Add `else` Block

To avoid the rule violation, add a terminating `else` block. The block can be empty.

```
int get_flag_1(void);
int get_flag_2(void);
void action_1(void);
void action_2(void);

void f1(void) {
    int flag_1 = get_flag_1(), flag_2 = get_flag_2();
    if(flag_1) {
```

```
        action_1();
    }
    else if(flag_2) {
        /* Non-compliant */
        action_2();
    }
    else {
        /* No statement required */
        /* ; is optional */
    }

}
```

Check Information

Group: Control Flow
Category: Required
AGC Category: Readability

See Also

`MISRA C:2012 Rule 16.5` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 16.1

All switch statements shall be well-formed

Description

Rule Definition

All switch statements shall be well-formed

Rationale

The syntax for switch statements in C is not particularly rigorous and can allow complex, unstructured behavior. This rule and other rules impose a simple consistent structure on the `switch` statement.

Polyspace Implementation

Following the MISRA specifications, the coding rules checker also raises a violation of rule 16.1 if a `switch` statement violates one of these rules: 16.2, 16.3, 16.4, 16.5 or 16.6.

Additional Message in Report

All messages in report file begin with "MISRA-C switch statements syntax normative restriction."

- Initializers shall not be used in switch clauses.
- The child statement of a switch shall be a compound statement.
- All switch clauses shall appear at the same level.
- A switch clause shall only contain switch labels and switch clauses, and no other code.
- A switch statement shall only contain switch labels and switch clauses, and no other code.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 15.3|MISRA C:2012 Rule 16.2|MISRA C:2012 Rule 16.3|MISRA C:2012 Rule 16.4|MISRA C:2012 Rule 16.5|MISRA C:2012 Rule 16.6|Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 16.2

A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement

Description

Rule Definition

A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement

Rationale

The C Standard permits placing a switch label (for instance, `case` or `default`) before any statement contained in the body of a switch statement. This flexibility can lead to unstructured code. To prevent unstructured code, make sure a switch label appears only at the outermost level of the body of a switch statement.

Additional Message in Report

All messages in report file begin with "MISRA-C switch statements syntax normative restriction."

- Initializers shall not be used in switch clauses.
- The child statement of a switch shall be a compound statement.
- All switch clauses shall appear at the same level.
- A switch clause shall only contain switch labels and switch clauses, and no other code.
- A switch statement shall only contain switch labels and switch clauses, and no other code.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 16.1` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 16.3

An unconditional break statement shall terminate every switch-clause

Description

Rule Definition

An unconditional break statement shall terminate every switch-clause

Rationale

A *switch-clause* is a case containing at least one statement. Two consecutive labels without an intervening statement is compliant with MISRA.

If you fail to end your switch-clauses with a break statement, then control flow "falls" into the next statement. This next statement can be another switch-clause, or the end of the switch. This behavior is sometimes intentional, but more often it is an error. If you add additional cases later, an unterminated switch-clause can cause problems.

Polyspace Implementation

Polyspace raises a warning for each noncompliant `case` clause.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 16.1` | `Check MISRA C:2012 (-misra3)`

Topics

"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 16.4

Every switch statement shall have a default label

Description

Rule Definition

Every switch statement shall have a default label

Rationale

The requirement for a `default` label is defensive programming. Even if your switch covers all possible values, there is no guarantee that the input takes one of these values. Statements following the `default` label take some appropriate action. If the `default` label requires no action, use comments to describe why there are no specific actions.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Switch Statement Without `default`

```
short func1(short xyz){

    switch(xyz){      /* Non-compliant - default label is required */
        case 0:
            ++xyz;
            break;
        case 1:
        case 2:
            break;
    }
    return xyz;
}
```

In this example, the switch statement does not include a `default` label, and is therefore noncompliant.

Correction — Add `default` With Error Flag

One possible correction is to use the `default` label to flag input errors. If your switch-clauses cover all expected input, then the default cases flags any input errors.

```
short func1(short xyz){

    switch(xyz){      /* Compliant */
        case 0:
            ++xyz;
            break;
        case 1:
        case 2:
```

```
            break;
        default:
            errorflag = 1;
            break;
    }
    if (errorflag == 1)
        return errorflag;
    else
        return xyz;
}
```

Switch Statement for Enumerated Inputs

```
enum Colors{
    RED, GREEN, BLUE
};

enum Colors func2(enum Colors color){
    enum Colors next;

    switch(color){      /* Non-compliant - default label is required */
        case RED:
            next = GREEN;
            break;
        case GREEN:
            next = BLUE;
            break;
        case BLUE:
            next = RED;
            break;
    }
    return next;
}
```

In this example, the switch statement does not include a `default` label, and is therefore noncompliant. Even though this switch statement handles all values of the enumeration, there is no guarantee that color takes one of the those values.

Correction — Add `default`

To be compliant, add the `default` label to the end of your switch. You can use this case to flag unexpected inputs.

```
enum Colors{
    RED, GREEN, BLUE, ERROR
};

enum Colors func2(enum Colors color){
    enum Colors next;

    switch(color){      /* Compliant */
        case RED:
            next = GREEN;
            break;
        case GREEN:
            next = BLUE;
            break;
        case BLUE:
```

```
            next = RED;
            break;
        default:
            next = ERROR;
            break;
    }

    return next;
}
```

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 2.1` | `MISRA C:2012 Rule 16.1` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 16.5

A default label shall appear as either the first or the last switch label of a switch statement

Description

Rule Definition

A default label shall appear as either the first or the last switch label of a switch statement.

Rationale

Using this rule, you can easily locate the `default` label within a `switch` statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Default Case in `switch` Statements

```
void foo(int var){

    switch(var){
        default:   /* Compliant - default is the first label */
        case 0:
            ++var;
            break;
        case 1:
        case 2:
            break;
    }

    switch(var){
        case 0:
            ++var;
            break;
        default:    /* Non-compliant - default is mixed with the case labels */
        case 1:
        case 2:
            break;
    }

    switch(var){
        case 0:
            ++var;
            break;
        case 1:
        case 2:
        default:     /* Compliant - default is the last label */
            break;
    }
```

```
    switch(var){
        case 0:
            ++var;
            break;
        case 1:
        case 2:
            break;
        default:      /* Compliant - default is the last label */
            var = 0;
            break;
    }
}
```

This example shows the same switch statement several times, each with `default` in a different place. As the first, third, and fourth switch statements show, `default` must be the first or last label. `default` can be part of a compound switch-clause (for instance, the third `switch` example), but it must be the last listed.

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 15.7` | `MISRA C:2012 Rule 16.1` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 16.6

Every switch statement shall have at least two switch-clauses

Description

Rule Definition

Every switch statement shall have at least two switch-clauses.

Rationale

A switch statement with a single path is redundant and can indicate a programming error.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 16.1` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 16.7

A switch-expression shall not have essentially Boolean type

Description

Rule Definition

A switch-expression shall not have essentially Boolean type

Rationale

The C Standard requires the controlling expression to a `switch` statement to have an integer type. Because C implements Boolean values with integer types, it is possible to have a Boolean expression control a `switch` statement. For controlling flow with Boolean types, an `if-else` construction is more appropriate.

Polyspace Implementation

The analysis recognizes the Boolean types, `bool` or `_Bool` (defined in `stdbool.h`)

You can also define types that are essentially Boolean using the option `Effective boolean types (-boolean-types)`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 17.1

The features of <stdarg.h> shall not be used

Description

Rule Definition

The features of <stdarg.h> shall not be used..

Rationale

The rule forbids use of `va_list`, `va_arg`, `va_start`, `va_end`, and `va_copy`.

You can use these features in ways where the behavior is not defined in the Standard. For instance:

- You invoke `va_start` in a function but do not invoke the corresponding `va_end` before the function block ends.
- You invoke `va_arg` in different functions on the same variable of type `va_list`.
- `va_arg` has the syntax `type va_arg (va_list ap, type)`.

  You invoke `va_arg` with a `type` that is incompatible with the actual type of the argument retrieved from `ap`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `va_start`, `va_list`, `va_arg`, and `va_end`

```
#include<stdarg.h>
void f2(int n, ...) {
    int i;
    double val;
    va_list vl;                          /* Non-compliant */

    va_start(vl, n);                     /* Non-compliant */

    for(i = 0; i < n; i++)
    {
        val = va_arg(vl, double);          /* Non-compliant */
    }

    va_end(vl);                          /* Non-compliant */
}
```

In this example, the rule is violated because `va_start`, `va_list`, `va_arg` and `va_end` are used.

Undefined Behavior of `va_arg`

```
#include <stdarg.h>
```

```
void h(va_list ap) {                    /* Non-compliant */
    double y;

    y = va_arg(ap, double );              /* Non-compliant */
}

void g(unsigned short n, ...) {
    unsigned int x;
    va_list ap;                         /* Non-compliant */

    va_start(ap, n);                    /* Non-compliant */
    x = va_arg(ap, unsigned int);       /* Non-compliant */

    h(ap);

    /* Undefined - ap is indeterminate because va_arg used in h () */
    x = va_arg(ap, unsigned int);       /* Non-compliant */

}

void f(void) {
    /* undefined - uint32_t:double type mismatch when g uses va_arg () */
    g(1, 2.0, 3.0);
}
```

In this example, `va_arg` is used on the same variable `ap` of type `va_list` in both functions `g` and `h`. In `g`, the second argument is `unsigned int` and in `h`, the second argument is `double`. This type mismatch causes undefined behavior.

Check Information

Group: Function
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 17.2

Functions shall not call themselves, either directly or indirectly

Description

Rule Definition

Functions shall not call themselves, either directly or indirectly.

Rationale

Variables local to a function are stored in the call stack. If a function calls itself directly or indirectly several times, the available stack space can be exceeded, causing serious failure. Unless the recursion is tightly controlled, it is difficult to determine the maximum stack space required.

Polyspace Implementation

The checker reports each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported.

You can calculate the total number of recursion cycles using the code complexity metric `Number of Recursions`.

Additional Message in Report

Message in Report: Function XX shall not call itself either directly or indirectly. Function XX is called indirectly by YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Direct and Indirect Recursion

```
void foo1( void ) {     /* Non-compliant - Indirect recursion foo1->foo2->foo1... */
    foo2();
    foo1();               /* Non-compliant - Direct recursion */
}

void foo2( void ) {
    foo1();
}
```

In this example, the rule is violated because of:

- Direct recursion `foo1` → `foo1`.
- Indirect recursion `foo1` → `foo2` → `foo1`.

Check Information

Group: Function
Category: Required
AGC Category: Required

See Also

`Number of Recursions` | `Number of Direct Recursions` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 17.3

A function shall not be declared implicitly

Description

Rule Definition

A function shall not be declared implicitly.

Rationale

An implicit declaration occurs when you call a function before declaring or defining it. When you declare a function explicitly before calling it, the compiler can match the argument and return types with the parameter types in the declaration. If an implicit declaration occurs, the compiler makes assumptions about the argument and return types. For instance, it assumes a return type of `int`. The assumptions might not agree with what you expect and cause undesired type conversions.

Additional Message in Report

Function 'XX' has no complete visible prototype at call.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Function Not Declared Before Call

```
#include <math.h>

extern double power3 (double val, int exponent);
int getChoice(void);

double func() {
    double res;
    int ch = getChoice();
    if(ch == 0) {
        res = power(2.0, 10);    /* Non-compliant */
    }
    else if( ch==1) {
        res = power2(2.0, 10);   /* Non-compliant */
    }
    else {
        res = power3(2.0, 10);   /* Compliant */
        return res;
    }
}

double power2 (double val, int exponent) {
    return (pow(val, exponent));
}
```

In this example, the rule is violated when a function that is not declared is called in the code. Even if a function definition exists later in the code, the rule violation occurs.

The rule is not violated when the function is declared before it is called in the code. If the function definition exists in another file and is available only during the link phase, you can declare the function in one of the following ways:

- Declare the function with the `extern` keyword in the current file.
- Declare the function in a header file and include the header file in the current file.

Check Information

Group: Function
Category: Mandatory
AGC Category: Mandatory

See Also

`MISRA C:2012 Rule 8.2` | `MISRA C:2012 Rule 8.4` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 17.4

All exit paths from a function with non-void return type shall have an explicit return statement with an expression

Description

Rule Definition

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Rationale

If a non-`void` function does not explicitly return a value but the calling function uses the return value, the behavior is undefined. To prevent this behavior:

1. You must provide `return` statements with an explicit expression.
2. You must ensure that during run time, at least one `return` statement executes.

Additional Message in Report

Missing return value for non-void function 'XX'.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Missing Return Statement Along Certain Execution Paths

```
int absolute(int v) {
    if(v < 0) {
        return v;
    }
}
```

In this example, the rule is violated because a `return` statement does not exist on all execution paths. If `v >= 0`, then the control returns to the calling function without an explicit return value.

Return Statement Without Explicit Expression

```
#define SIZE 10
int table[SIZE];

unsigned short lookup(unsigned short v) {
    if((v < 0) || (v > SIZE)) {
        return;
    }
    return table[v];
}
```

In this example, the rule is violated because the `return` statement in the `if` block does not have an explicit expression.

Check Information

Group: Function
Category: Mandatory
AGC Category: Mandatory

See Also

`MISRA C:2012 Rule 15.5` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 17.5

The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements

Description

Rule Definition

The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.

Rationale

If you use an array declarator for a function parameter instead of a pointer, the function interface is clearer because you can state the minimum expected array size. If you do not state a size, the expectation is that the function can handle an array of any size. In such cases, the size value is typically another parameter of the function, or the array is terminated with a sentinel value.

However, it is legal in C to specify an array size but pass an array of smaller size. This rule prevents you from passing an array of size smaller than the size you declared.

Additional Message in Report

The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.

The argument type has *`actual_size`* elements whereas the parameter type expects *`expected_size`* elements.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Incorrect Array Size Passed to Function

```
void func(int arr[4]);

int main() {
    int arrSmall[3] = {1,2,3};
    int arr[4] = {1,2,3,4};
    int arrLarge[5] ={1,2,3,4,5};

    func(arrSmall);      /* Non-compliant */
    func(arr);           /* Compliant */
    func(arrLarge);      /* Compliant */

    return 0;
}
```

In this example, the rule is violated when `arrSmall`, which has size 3, is passed to `func`, which expects at least 4 elements.

Check Information

Group: Functions
Category: Advisory
AGC Category: Readability

See Also

`Check MISRA C:2012 (-misra3)|MISRA C:2012 Rule 17.6`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Rule 17.6

The declaration of an array parameter shall not contain the static keyword between the [ ]

Description

Rule Definition

The declaration of an array parameter shall not contain the static keyword between the [ ].

Rationale

If you use the `static` keyword within `[]` for an array parameter of a function, you can inform a C99 compiler that the array contains a minimum number of elements. The compiler can use this information to generate efficient code for certain processors. However, in your function call, if you provide less than the specified minimum number, the behavior is not defined.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `static` Keyword Within `[]` in Array Parameter

```
extern int arr1[20];
extern int arr2[10];

/* Non-compliant: static keyword used in array declarator */
unsigned int total (unsigned int n, unsigned int arr[static 20]) {
    unsigned int i;
    unsigned int sum = 0;

    for (i=0U; i < n; i++) {
        sum+= arr[i];
    }

    return sum;
}

void func (void) {
    int res, res2;
    res = total (10U, arr1);  /* Non-compliant - behavior not defined */
    res2 = total (20U, arr2); /* Non-compliant, even if behavior is defined */
}
```

In this example, the rule is violated when the `static` keyword is used within `[]` in the array parameter of function `total`. Even if you call `total` with array arguments where the behavior is well-defined, the rule violation occurs.

Check Information

Group: Function

Category: Mandatory
AGC Category: Mandatory

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 17.7

The value returned by a function having non-void return type shall be used

Description

Rule Definition

The value returned by a function having non-void return type shall be used.

Rationale

You can unintentionally call a function with a non-`void` return type but not use the return value. Because the compiler allows the call, you might not catch the omission. This rule forbids calls to a non-`void` function where the return value is not used. If you do not intend to use the return value of a function, explicitly cast the return value to `void`.

Polyspace Implementation

The checker flags functions with non-`void` return if the return value is not used or not explicitly cast to a `void` type.

The checker does not flag the functions `memcpy`, `memset`, `memmove`, `strcpy`, `strncpy`, `strcat`, `strncat` because these functions simply return a pointer to their first arguments.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Used and Unused Return Values

```
unsigned int cutOff(unsigned int val) {
    if (val > 10 && val < 100) {
        return val;
    }
    else {
        return 0;
    }
}

unsigned int getVal(void);

void func2(void) {
    unsigned int val = getVal(), res;
    cutOff(val);           /* Non-compliant */
    res = cutOff(val);     /* Compliant */
    (void)cutOff(val);     /* Compliant */
}
```

In this example, the rule is violated when the return value of `cutOff` is not used subsequently.

The rule is not violated when the return value is:

- Assigned to another variable.
- Explicitly cast to `void`.

Check Information

Group: Function
Category: Required
AGC Category: Readability

See Also

`MISRA C:2012 Rule 2.2` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 17.8

A function parameter should not be modified

Description

Rule Definition

A function parameter should not be modified.

Rationale

When you modify a parameter, the function argument corresponding to the parameter is not modified. However, you or another programmer unfamiliar with C can expect by mistake that the argument is also modified when you modify the parameter.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Function Parameter Modified

```
int input(void);

void func(int param1, int* param2) {

    param1 = input();   /* Non-compliant */
    *param2 = input();  /* Compliant */
}
```

In this example, the rule is violated when the parameter `param1` is modified.

The rule is not violated when the parameter is a pointer `param2` and `*param2` is modified.

Check Information

Group: Functions
Category: Advisory
AGC Category: Readability

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Rule 18.1

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

Description

Rule Definition

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

Rationale

Using an invalid array subscript can lead to erroneous behavior of the program. Run-time derived array subscripts are especially troublesome because they cannot be easily checked by manual review or static analysis.

The C Standard defines the creation of a pointer to one beyond the end of the array. The rule permits the C Standard. Dereferencing a pointer to one beyond the end of an array causes undefined behavior and is noncompliant.

Polyspace Implementation

Polyspace flags this rule during the analysis as:

- Bug Finder — `Array access out-of-bounds` and `Pointer access out-of-bounds`.
- Code Prover — `Illegally dereferenced pointer` and `Out of bounds array index`.

Bug Finder and Code Prover check this rule differently and can show different results for this rule. In Code Prover, you can also see a difference in results based on your choice for the option . See “Check for Coding Standard Violations”.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Dir 4.1` | `MISRA C:2012 Rule 18.4` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 18.2

Subtraction between pointers shall only be applied to pointers that address elements of the same array

Description

Rule Definition

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Rationale

This rule applies to expressions of the form `pointer_expression1 - pointer_expression2`. The behavior is undefined if `pointer_expression1` and `pointer_expression2`:

- Do not point to elements of the same array,
- Or do not point to the element one beyond the end of the array.

Polyspace Implementation

This rule is raised whenever the analysis detects a `Subtraction or comparison between pointers to different arrays`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Subtracting Pointers

```
#include <stddef.h>

void f1 (int32_t *ptr)
{
    int32_t a1[10];
    int32_t a2[10];
    int32_t *p1 = &a1[ 1];
    int32_t *p2 = &a2[10];
    ptrdiff_t diff1, diff2, diff3;

    diff1 =  p1 - a1;   // Compliant
    diff2 =  p2 - a2;   // Compliant
    diff3 =  p1 - p2;   // Non-compliant
}
```

In this example, the three subtraction expressions show the difference between compliant and noncompliant pointer subtractions. The `diff1` and `diff2` subtractions are compliant because the pointers point to the same array. The `diff3` subtraction is not compliant because `p1` and `p2` point to different arrays.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Dir 4.1` | `MISRA C:2012 Rule 18.4` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 18.3

The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object

Description

Rule Definition

The relational operators >, >=, <, and <= shall not be applied to objects of pointer type except where they point into the same object.

Rationale

If two pointers do not point to the same object, comparisons between the pointers produces undefined behavior.

You can address the element beyond the end of an array, but you cannot access this element.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Pointer and Array Comparisons

```
void f1(void){
    int arr1[10];
    int arr2[10];
    int *ptr1 = arr1;

    if(ptr1 < arr2){}    /* Non-compliant */
    if(ptr1 < arr1){}    /* Compliant */
}
```

In this example, `ptr1` is a pointer to `arr1`. To be compliant with rule 18.3, you can compare only `ptr1` with `arr1`. Therefore, the comparison between `ptr1` and `arr2` is noncompliant.

Structure Comparisons

```
struct limits{
  int lower_bound;
  int upper_bound;
};

void func2(void){
    struct limits lim_1 = { 2, 5 };
    struct limits lim_2 = { 10, 5 };

    if(&lim_1.lower_bound <= &lim_2.upper_bound){}  /* Non-compliant *
    if(&lim_1.lower_bound <= &lim_1.upper_bound){}  /* Compliant */
}
```

This example defines two `limits` structures, `lim1` and `lim2`, and compares the elements. To be compliant with rule 18.3, you can compare only the structure elements within a structure. The first comparison compares the `lower_bound` of `lim1` and the `upper_bound` of `lim2`. This comparison is noncompliant because the `lim_1.lower_bound` and `lim_2.upper_bound` are elements of two different structures.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Dir 4.1|Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 18.4

The +, -, += and -= operators should not be applied to an expression of pointer type

Description

Rule Definition

The +, -, += and -= operators should not be applied to an expression of pointer type.

Rationale

The preferred form of pointer arithmetic is using the array subscript syntax `ptr[expr]`. This syntax is clear and less prone to error than pointer manipulation. With pointer manipulation, any explicitly calculated pointer value has the potential to access unintended or invalid memory addresses. Array indexing can also access unintended or invalid memory, but it is easier to review.

To a new C programmer, the expression `ptr+1` can be mistakenly interpreted as one plus the address of `ptr`. However, the new memory address depends on the size, in bytes, of the pointer's target. This confusion can lead to unexpected behavior.

When used with caution, pointer manipulation using ++ can be more natural (for instance, sequentially accessing locations during a memory test).

Polyspace Implementation

Polyspace flags operations on pointers, for example, `Pointer + Integer`, `Integer + Pointer`, `Pointer - Integer`.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Pointers and Array Expressions

```
void fun1(void){
    unsigned char arr[10];
    unsigned char *ptr;
    unsigned char index = 0U;

    index = index + 1U;   /* Compliant - rule only applies to pointers */

    arr[index] = 0U;      /* Compliant */
    ptr = &arr[5];        /* Compliant */
    ptr = arr;
    ptr++;                /* Compliant - increment operator not + */
    *(ptr + 5) = 0U;      /* Non-compliant */
    ptr[5] = 0U;          /* Compliant */
}
```

This example shows various operations with pointers and arrays. The only operation in this example that is noncompliant is using the + operator directly with a pointer (line 12).

Adding Array Elements Inside a `for` Loop

```
void fun2(void){
    unsigned char array_2_2[2][2] = {{1U, 2U}, {4U, 5U}};
    unsigned char i = 0U;
    unsigned char j = 0U;
    unsigned char sum = 0U;

    for(i = 0u; i < 2U; i++){
        unsigned char *row = array_2_2[ i ];

        for(j = 0u; j < 2U; j++){
            sum += row[ j ];                    /* Compliant */
        }
    }
}
```

In this example, the second `for` loop uses the array pointer `row` in an arithmetic expression. However, this usage is compliant because it uses the array index form.

Pointers and Array Expressions

```
void fun3(unsigned char *ptr1, unsigned char ptr2[ ]){
    ptr1++;               /* Compliant */
    ptr1 = ptr1 - 5;      /* Non-compliant */
    ptr1 -= 5;            /* Non-compliant */
    ptr1[2] = 0U;         /* Compliant */

    ptr2++;               /* Compliant */
    ptr2 = ptr2 + 3;      /* Non-compliant */
    ptr2 += 3;            /* Non-compliant */
    ptr2[3] = 0U;         /* Compliant */
}
```

This example shows the offending operators used on pointers and arrays. Notice that the same types of expressions are compliant and noncompliant for both pointers and arrays.

If `ptr1` does not point to an array with at least six elements, and `ptr2` does not point to an array with at least 4 elements, this example violates rule 18.1.

Check Information

Group: Pointers and Arrays
Category: Advisory
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 18.1` | `MISRA C:2012 Rule 18.2` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 18.5

Declarations should contain no more than two levels of pointer nesting

Description

Rule Definition

Declarations should contain no more than two levels of pointer nesting.

Rationale

The use of more than two levels of pointer nesting can seriously impair the ability to understand the behavior of the code. Avoid this usage.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Pointer Nesting

```
typedef char *INTPTR;

void function(char ** arrPar[ ])    /* Non-compliant - 3 levels */
{
    char   **  obj2;            /* Compliant */
    char   *** obj3;            /* Non-compliant */
    INTPTR *   obj4;            /* Compliant */
    INTPTR * const * const obj5;    /* Non-compliant */
    char   ** arr[10];            /* Compliant */
    char   ** (*parr)[10];        /* Compliant */
    char   *  (**pparr)[10];        /* Compliant */
}

struct s{
    char *   s1;                /* Compliant */
    char **  s2;                /* Compliant */
    char *** s3;                /* Non-compliant */
};

struct s *   ps1;            /* Compliant */
struct s **  ps2;            /* Compliant */
struct s *** ps3;            /* Non-compliant */

char **  (  *pfunc1)(void);        /* Compliant */
char **  ( **pfunc2)(void);        /* Compliant */
char **  (***pfunc3)(void);        /* Non-compliant */
char *** ( **pfunc4)(void);        /* Non-compliant */
```

This example shows various pointer declarations and nesting levels. Any pointer with more than two levels of nesting is considered noncompliant.

Check Information

Group: Pointers and Arrays
Category: Advisory
AGC Category: Readability

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 18.6

The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist

Description

Rule Definition

The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.

Rationale

The address of an object becomes indeterminate when the lifetime of that object expires. Any use of an indeterminate address results in undefined behavior.

Polyspace Implementation

Polyspace flags a violation when assigning an address to a global variable, returning a local variable address, or returning a parameter address.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Address of Local Variables

```
char *func(void){
    char local_auto;
    return &local_auto  /* Non-compliant
                         * &local_auto is indeterminate */
}
```

In this example, because `local_auto` is a local variable, after the function returns, the address of `local_auto` is indeterminate.

Copying Pointer Addresses to Local Variables

```
char *sp;

void f(unsigned short u){
    g(&u);
}

void g(unsigned short *p){
    sp = p;  /* Non-compliant
              * the parameter u from f is copied to static sp */
}

void h(void){
```

```
    static unsigned short *q;

    unsigned short x =0u;
    q = &x;  /* Non-compliant -
              * &x stored in object with greater lifetime */
}
```

In this example, the function g stores a copy of its pointer parameter p. If p always points to an object with static storage duration, then the code is compliant with this rule. However, in this example, p points to an object with automatic storage duration. In such a case, copying the parameter p is noncompliant.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 18.7

Flexible array members shall not be declared

Description

Rule Definition

Flexible array members shall not be declared.

Rationale

Flexible array members are usually used with dynamic memory allocation. Dynamic memory allocation is banned by Directive 4.12 and Rule 21.3 on page 5-187.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 21.3|Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 18.8

Variable-length array types shall not be used

Description

Rule Definition

Variable-length array types shall not be used.

Rationale

When the size of an array declared in a block or function prototype is not an integer constant expression, you specify variable array types. Variable array types are typically implemented as a variable size object stored on the stack. Using variable type arrays can make it impossible to determine statistically the amount of memory for the stack requires.

If the size of a variable-length array is negative or zero, the behavior is undefined.

If a variable-length array must be compatible with another array type, then the size of the array types must be identical and positive integers. If your array does not meet these requirements, the behavior is undefined.

If you use a variable-length array type in a `sizeof`, it is uncertain if the array size is evaluated or not.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 13.6` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 19.1

An object shall not be assigned or copied to an overlapping object

Description

Rule Definition

An object shall not be assigned or copied to an overlapping object.

Rationale

When you assign an object to another object with overlapping memory, the behavior is undefined. The exceptions are:

- You assign an object to another object with exactly overlapping memory and compatible type.
- You copy one object to another using `memmove`.

Additional Message in Report

- An object shall not be assigned or copied to an overlapping object.
- Destination and source of XX overlap, the behavior is undefined.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Assignment of Union Members

```
void func (void) {
    union {
        short i;
        int j;
    } a = {0}, b = {1};

    a.j = a.i;   /* Non-compliant */
    a = b;       /* Compliant */
}
```

In this example, the rule is violated when `a.i` is assigned to `a.j` because the two variables have overlapping regions of memory.

Assignment of Array Segments

```
#include <string.h>

int arr[10];

void func(void) {
    memcpy (&arr[5], &arr[4], 2u * sizeof(arr[0]));    /* Non-compliant */
    memcpy (&arr[5], &arr[4], sizeof(arr[0]));         /* Compliant */
```

```
    memcpy (&arr[1], &arr[4], 2u * sizeof(arr[0]));    /* Compliant */
}
```

In this example, memory equal to twice `sizeof(arr[0])` is the memory space taken up by two array elements. If that memory space begins from `&a[4]` and `&a[5]`, the two memory regions overlap. The rule is violated when the `memcpy` function is used to copy the contents of these two overlapping memory regions.

Check Information

Group: Overlapping Storage
Category: Mandatory
AGC Category: Mandatory

See Also

`MISRA C:2012 Rule 19.2` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 19.2

The union keyword should not be used

Description

Rule Definition

The union keyword should not be used.

Rationale

If you write to a union member and read the same union member, the behavior is well-defined. But if you read a different member, the behavior depends on the relative sizes of the members. For instance:

- If you read a union member with wider memory size, the value you read is unspecified.
- Otherwise, the value is implementation-dependent.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Possible Problems with `union` Keyword

```
unsigned int zext(unsigned int s)
{
    union                 /* Non-compliant */
    {
        unsigned int ul;
        unsigned short us;
    } tmp;

    tmp.us = s;
    return tmp.ul;        /* Unspecified value */
}
```

In this example, the 16-bit `short` field `tmp.us` is written but the wider 32-bit `int` field `tmp.ul` is read. Using the `union` keyword can cause such unspecified behavior. Therefore, the rule forbids using the `union` keyword.

Check Information

Group: Overlapping Storage
Category: Advisory
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 19.1` | `Check MISRA C:2012 (-misra3)`

Topics
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 2.1

A project shall not contain unreachable code

Description

Rule Definition

A project shall not contain unreachable code.

Rationale

Unless a program exhibits any undefined behavior, unreachable code cannot execute. The unreachable code cannot affect the program output. The presence of unreachable code can indicate an error in the program logic. Unreachable code that the compiler does not remove wastes resources, for example:

- It occupies space in the target machine memory.
- Its presence can cause a compiler to select longer, slower jump instructions when transferring control around the unreachable code.
- Within a loop, it can prevent the entire loop from residing in an instruction cache.

Polyspace Implementation

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

The Code Prover run-time check for unreachable code shows more cases than the MISRA checker for rule 2.1. See also `Unreachable code`. The run-time check performs a more exhaustive analysis. In the process, the check can show some instances that are not strictly unreachable code but unreachable only in the context of the analysis. For instance, in the following code, the run-time check shows a potential division by zero in the first line and then removes the zero value of `flag` for the rest of the analysis. Therefore, it considers the `if` block unreachable.

```
val=1.0/flag;
if(!flag) {}
```

The MISRA checker is designed to prevent these kinds of results.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Code Following `return` Statement

```
enum light { red, amber, red_amber, green };

enum light next_light ( enum light color )
{
    enum light res;
```

```
    switch ( color )
    {
    case red:
        res = red_amber;
        break;
    case red_amber:
        res = green;
        break;
    case green:
        res = amber;
        break;
    case amber:
        res = red;
        break;
    default:
    {
        error_handler ();
        break;
    }
    }

    res = color;
    return res;
    res = color;     /* Non-compliant */
}
```

In this example, the rule is violated because there is an unreachable operation following the `return` statement.

Check Information

Group: Unused Code
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 14.3` | `MISRA C:2012 Rule 16.4` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 2.2

There shall be no dead code

Description

Rule Definition

There shall be no dead code.

Rationale

If an operation is reachable but removing the operation does not affect program behavior, the operation constitutes dead code.

The presence of dead code can indicate an error in the program logic. Because a compiler can remove dead code, its presence can cause confusion for code reviewers.

Operations involving language extensions such as `__asm ( "NOP" );` are not considered dead code.

Polyspace Implementation

Polyspace Bug Finder detects useless write operations during analysis.

Polyspace Code Prover does not detect useless write operations. For instance, if you assign a value to a local variable but do not read it later, Polyspace Code Prover does not detect this useless assignment. Use Polyspace Bug Finder to detect such useless write operations.

In Code Prover, you can also see a difference in results based on your choice for the option . See “Check for Coding Standard Violations”.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Redundant Operations

```
extern volatile unsigned int v;
extern char *p;

void f ( void ) {
    unsigned int x;


    ( void ) v;      /* Compliant - Exception*/
    ( int ) v;       /* Non-compliant  */
    v >> 3;          /* Non-compliant  */

    x = 3;           /* Non-compliant - Detected in Bug Finder only */
```

```
    *p++;            /* Non-compliant  */
    ( *p )++;        /* Compliant  */
}
```

In this example, the rule is violated when an operation is performed on a variable, but the result of that operation is not used. For instance,

- The operations `(int)` and `>>` on the variable `v` are redundant because the results are not used.
- The operation `=` is redundant because the local variable `x` is not read after the operation.
- The operation `*` on `p++` is redundant because the result is not used.

The rule is not violated when:

- A variable is cast to `void`. The cast indicates that you are intentionally not using the value.
- The result of an operation is used. For instance, the operation `*` on `p` is not redundant, because `*p` is incremented.

Redundant Function Call

```
void g ( void ) {
               /* Compliant  */
}

void h ( void) {
     g( );     /* Non-compliant */
}
```

In this example, `g` is an empty function. Though the function itself does not violate the rule, a call to the function violates the rule.

Check Information

Group: Unused Code
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 17.7|Write without a further read|Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 2.3

A project should not contain unused type declarations

Description

Rule Definition

A project should not contain unused type declarations.

Rationale

If a type is declared but not used, a reviewer does not know if the type is redundant or if it is unused by mistake.

Additional Message in Report

A project should not contain unused type declarations: type XX is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Unused Local Type

```
signed short unusedType (void){

    typedef signed short myType;   /* Non-compliant */
    return 67;

}

signed short usedType (void){

    typedef signed short myType;  /* Compliant */
    myType tempVar = 67;
    return tempVar;

}
```

In this example, in function `unusedType`, the `typedef` statement defines a new local type `myType`. However, this type is never used in the function. Therefore, the rule is violated.

The rule is not violated in the function `usedType` because the new type `myType` is used.

Check Information

Group: Unused Code
Category: Advisory
AGC Category: Readability

See Also

`MISRA C:2012 Rule 2.4` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 2.4

A project should not contain unused tag declarations

Description

Rule Definition

A project should not contain unused tag declarations.

Rationale

If a tag is declared but not used, a reviewer does not know if the tag is redundant or if it is unused by mistake.

Additional Message in Report

A project should not contain unused tag declarations: tag *`tag_name`* is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Tag Defined in Function but Not Used

```
void unusedTag ( void )
{
    enum state1 { S_init, S_run, S_sleep };    /* Non-compliant  */
}

void usedTag ( void )
{
    enum state2 { S_init, S_run, S_sleep };    /* Compliant  */
    enum state2 my_State = S_init;
}
```

In this example, in the function `unusedTag`, the tag `state1` is defined but not used. Therefore, the rule is violated.

Tag Used in `typedef` Only

```
typedef struct record_t        /* Non-compliant  */
{
    unsigned short key;
    unsigned short val;
} record1_t;


typedef struct               /* Compliant */
{
    unsigned short key;
    unsigned short val;
```

```
} record2_t;

record1_t myRecord1_t;
record2_t myRecord2_t;
```

In this example, the tag `record_t` appears only in the `typedef` of `record1_t`. In the rest of the translation unit, the type `record1_t` is used. Therefore, the rule is violated.

Check Information

Group: Unused Code
Category: Advisory
AGC Category: Readability

See Also

`MISRA C:2012 Rule 2.3 | Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 2.5

A project should not contain unused macro declarations

Description

Rule Definition

A project should not contain unused macro declarations.

Rationale

If a macro is declared but not used, a reviewer does not know if the macro is redundant or if it is unused by mistake.

Additional Message in Report

A project should not contain unused macro declarations: macro *`macro_name`* is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Unused Macro Definition

```
void use_macro (void)
{
    #define SIZE 4
    #define DATA 3

    use_int16(SIZE);
}
```

In this example, the macro `DATA` is never used in the `use_macro` function.

Check Information

Group: Unused Code
Category: Advisory
AGC Category: Readability

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 2.6

A function should not contain unused label declarations

Description

Rule Definition

A function should not contain unused label declarations.

Rationale

If you declare a label but do not use it, it is not clear to a reviewer of your code if the label is redundant or unused by mistake.

Additional Message in Report

A function should not contain unused label declarations.

Label *`label_name`* is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Unused Label Declarations

```
void use_var(signed short);

void unused_label ( void )
{
    signed short x = 6;

label1:                          /* Non-compliant - label1 not used */
    use_var ( x );
}

void used_label ( void )
{
    signed short x = 6;

    for (int i=0; i < 5; i++) {
        if ( i==2 ) goto label1;
    }

label1:                         /* Compliant - label1 used */
    use_var ( x );
}
```

In this example, the rule is violated when the label `label1` in function `unused_label` is not used.

Check Information

Group: Unused code
Category: Advisory
AGC Category: Readability

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Rule 2.7

There should be no unused parameters in functions

Description

Rule Definition

There should be no unused parameters in functions.

Rationale

If a parameter is unused, it is possible that the implementation of the function does not match its specifications. This rule can highlight such mismatches.

Additional Message in Report

There should be no unused parameters in functions.

Parameter *`parameter_name`* is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Unused Function Parameters

```
double func(int param1, int* param2) {
    return (param1/2.0);
}
```

In this example, the rule is violated because the parameter `param2` is not used.

Check Information

Group: Unused code
Category: Advisory
AGC Category: Readability

See Also

`Check MISRA C:2012 (-misra3)` | `Unused parameter`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Rule 20.1

#include directives should only be preceded by preprocessor directives or comments

Description

Rule Definition

#include directives should only be preceded by preprocessor directives or comments.

Rationale

For better code readability, group all `#include` directives in a file at the top of the file. Undefined behavior can occur if you use `#include` to include a standard header file within a declaration or definition, or if you use part of the Standard Library before including the related standard header files.

Polyspace Implementation

Polyspace flags text that precedes a `#include` directive. Polyspace ignores preprocessor directives, comments, spaces, or "new lines".

Additional Message in Report

#include directives should only be preceded by preprocessor directives or comments.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Advisory
AGC Category: Advisory

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.10

The # and ## preprocessor operators should not be used

Description

Rule Definition

The # and ## preprocessor operators should not be used.

Rationale

The order of evaluation associated with multiple #, multiple ##, or a mix of # and ## preprocessor operators is unspecified. In some cases, it is therefore not possible to predict the result of macro expansion.

The use of ## can result in obscured code.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Advisory
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 1.3|MISRA C:2012 Rule 20.11|Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.11

A macro parameter immediately following a # operator shall not immediately be followed by a ## operator

Description

Rule Definition

A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.

Rationale

The order of evaluation associated with multiple #, multiple ##, or a mix of # and ## preprocessor operators, is unspecified. Rule 20.10 discourages the use of # and ##. The result of a # operator is a string literal. It is extremely unlikely that pasting this result to any other preprocessing token results in a valid token.

Additional Message in Report

The ## preprocessor operator shall not follow a macro parameter following a # preprocessor operator.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of # and

```
#define A( x )    #x             /* Compliant */
#define B( x, y ) x ## y         /* Compliant */
#define C( x, y ) #x ## y     /* Non-compliant */
```

In this example, you can see three uses of the # and ## operators. You can use these preprocessing operators alone (line 1 and line 2), but using # then ## is noncompliant (line 3).

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 20.10|Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”

“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.12

A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators

Description

Rule Definition

A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.

Rationale

The parameter to # or ## is not expanded prior to being used. The same parameter appearing elsewhere in the replacement text is expanded. If the macro parameter is itself subject to macro replacement, its use in mixed contexts within a macro replacement might not meet developer expectations.

Additional Message in Report

Expanded macro parameter *`param1`* is also an operand of *`op`* operator.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.13

A line whose first token is # shall be a valid preprocessing directive

Description

Rule Definition

A line whose first token is # shall be a valid preprocessing directive

Rationale

You typically use a preprocessing directive to conditionally exclude source code until a corresponding `#else`, `#elif`, or `#endif` directive is encountered. If your compiler does not detect a preprocessing directive because it is malformed or invalid, you can end up excluding more code than you intended.

If all preprocessing directives are syntactically valid, even in excluded code, this unintended code exclusion cannot happen.

Additional Message in Report

Directive is not syntactically meaningful.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.14

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related

Description

Rule Definition

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related.

Rationale

When conditional compilation directives include or exclude blocks of code and are spread over multiple files, confusion arises. If you terminate an `#if` directive within the same file, you reduce the visual complexity of the code and the chances of an error.

If you terminate `#if` directives within the same file, you can use `#if` directives in included files

Additional Message in Report

- '#else' not within a conditional.
- '#elseif' not within a conditional.
- '#endif' not within a conditional.

  Unterminated conditional directive.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.2

The ', " or \ characters and the /* or // character sequences shall not occur in a header file name

Description

Rule Definition

The ', " or \ characters and the / or // character sequences shall not occur in a header file name*.

Rationale

The program's behavior is undefined if:

- You use `'`, `"`, `\`, `/*` or `//` between `<` `>` delimiters in a header name preprocessing token.
- You use `'`, `\`, `/*` or `//` between `"` delimiters in a header name preprocessing token.

Although `\` results in undefined behavior, many implementations accept `/` in its place.

Polyspace Implementation

Polyspace flags the characters `'`, `"`, `\`, `/*` or `//` between `<` and `>` in `#include <filename>`.

Polyspace flags the characters `'`, `\`, `/*` or `//` between `"` and `"` in `#include "filename"`.

Additional Message in Report

The ', "or \ characters and the /* or // character sequences shall not occur in a header file name.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 20.3

The #include directive shall be followed by either a <filename> or "filename" sequence

Description

Rule Definition

The #include directive shall be followed by either a <filename> or "filename" sequence.

Rationale

This rule applies only after macro replacement.

The behavior is undefined if an `#include` directive does not use one of the following forms:

- `#include <filename>`
- `#include "filename"`

Additional Message in Report

- '#include' expects "FILENAME" or <FILENAME>
- '#include_next' expects "FILENAME" or <FILENAME>
- '#include' does not expect string concatenation.
- '#include_next' does not expect string concatenation.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 20.4

A macro shall not be defined with the same name as a keyword

Description

Rule Definition

A macro shall not be defined with the same name as a keyword.

Rationale

Using macros to change the meaning of keywords can be confusing. The behavior is undefined if you include a standard header while a macro is defined with the same name as a keyword.

Additional Message in Report

- The macro *`macro_name`* shall not be redefined.
- The macro *`macro_name`* shall not be undefined.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Redefining `int` keyword

```
#define int some_other_type
           /* Non-compliant - int keyword behavior altered */
#include <stdlib.h>
...
```

In this example, the `#define` violates Rule 20.4 because it alters the behavior of the `int` keyword. The inclusion of the standard header results in undefined behavior.

Correction — Rename keyword

One possible correction is to use a different keyword:

```
#define int_mine some_other_type
#include <stdlib.h>
...
```

Redefining keywords versus statements

```
#define while(E) for ( ; (E) ; )  /* Non-compliant - while redefined*/
#define unless(E) if ( !(E) )     /* Compliant*/

#define seq(S1, S2) do{ S1; S2;} while(false)  /* Compliant*/
#define compound(S) {S;}                       /* Compliant*/
...
```

In this example, it is noncompliant to redefine the keyword `while`, but it is compliant to define a macro that expands to statements.

Redefining keywords in different standards

```
#define inline
```

In this example, redefining `inline` is compliant in C90, but not in C99 because `inline` is not a keyword in C90.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)|MISRA C:2012 Rule 21.1`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.5

#undef should not be used

Description

Rule Definition

#undef should not be used.

Rationale

`#undef` can make the software unclear which macros exist at a particular point within a translation unit.

Additional Message in Report

#undef shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Advisory
AGC Category: Readability

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.6

Tokens that look like a preprocessing directive shall not occur within a macro argument

Description

Rule Definition

Tokens that look like a preprocessing directive shall not occur within a macro argument.

Rationale

An argument containing sequences of tokens that otherwise act as preprocessing directives leads to undefined behavior.

Polyspace Implementation

Polyspace looks for the # character in a macro arguments (outside a string or character constant).

Additional Message in Report

Macro argument shall not look like a preprocessing directive.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Macro Expansion Causing Non-Compliance

```
#define M( A ) printf ( #A )

#include <stdio.h>

void foo(void){
    M(
#ifdef SW         /* Non-compliant */
    "Message 1"
#else
    "Message 2"   /* Compliant - SW not defined */
#endif            /* Non-compliant */
    );
}
```

This example shows a macro definition and the macro usage. `#ifdef SW` and `#endif` are noncompliant because they look like a preprocessing directive. Polyspace does not flag `#else "Message 2"` because after macro expansion, Polyspace knows `SW` is not defined. The expanded macro is `printf ("\"Message 2\"");`

Check Information

Group: Preprocessing Directives

Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.7

Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses

Description

Rule Definition

Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.

Rationale

If you do not use parentheses, then it is possible that operator precedence does not give the results that you want when macro substitution occurs.

If you are not using a macro parameter as an expression, then the parentheses are not necessary because no operators are involved in the macro.

Additional Message in Report

Expanded macro parameter *`param`* shall be enclosed in parentheses.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Macro Expressions

```
#define mac1(x, y) (x * y)
#define mac2(x, y) ((x) * (y))

void foo(void){
    int r;

    r = mac1(1 + 2, 3 + 4);       /* Non-compliant */
    r = mac1((1 + 2), (3 + 4));   /* Compliant */

    r = mac2(1 + 2, 3 + 4);       /* Compliant */
}
```

In this example, `mac1` and `mac2` are two defined macro expressions. The definition of `mac1` does not enclose the arguments in parentheses. In line 7, the macro expands to `r = (1 + 2 * 3 + 4);` This expression can be `(1 + (2 * 3) + 4)` or `(1 + 2) * (3 + 4)`. However, without parentheses, the program does not know the intended expression. Line 8 uses parentheses, so the line expands to `(1 + 2) * (3 + 4)`. This macro expression is compliant.

The definition of `mac2` does enclose the argument in parentheses. Line 10 (the same macro arguments in line 7) expands to `(1 + 2) * (3 + 4)`. This macro and macro expression are compliant.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Dir 4.9` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.8

The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1

Description

Rule Definition

The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1.

Rationale

Strong typing requires that conditional inclusion preprocessing directives, `#if` or `#elif`, have a controlling expression that evaluates to a Boolean value.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 14.4|Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 20.9

All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation

Description

Rule Definition

All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation.

Rationale

If attempt to use a macro identifier in a preprocessing directive, and you have not defined that identifier, then the preprocessor assumes that it has a value of zero. This value might not meet developer expectations.

Additional Message in Report

`Identifier` is not defined.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Macro Identifiers

```
#if M == 0                    /* Non-compliant - Not defined */
#endif

#if defined (M)               /* Compliant - M is not evaluate */
#if M == 0                    /* Compliant - M is known to be defined */
#endif
#endif

#if defined (M) && (M == 0)  /* Compliant
                              * if M defined, M evaluated in ( M == 0 ) */
#endif
```

This example shows various uses of `M` in preprocessing directives. The second and third `#if` clauses check to see if the software defines `M` before evaluating `M`. The first `#if` clause does not check to see if `M` is defined, and because `M` is not defined, the statement is noncompliant.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Dir 4.9|Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 21.1

#define and #undef shall not be used on a reserved identifier or reserved macro name

Description

Rule Definition

#define and #undef shall not be used on a reserved identifier or reserved macro name.

Rationale

Reserved identifiers and reserved macro names are intended for use by the implementation. Removing or changing the meaning of a reserved macro can result in undefined behavior. This rule applies to the following:

- Identifiers or macro names beginning with an underscore
- Identifiers in file scope described in the C Standard Library
- Macro names described in the C Standard Library as being defined in a standard header

The rule checker can flag different identifiers or macros depending on the version of the C standard used in the analysis. See `C standard version (-c-version)`. For instance, if you run a C99 analysis, the reserved identifiers and macros are defined in the ISO/IEC 9899:1999 standard, Section 7, "Library".

Additional Message in Report

- The macro *`macro_name`* shall not be redefined.
- The macro *`macro_name`* shall not be undefined.
- The macro *`macro_name`* shall not be defined.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Defining or Undefining Reserved Identifiers

```
#undef __LINE__                /* Non-compliant - begins with _ */
#define _Guard_H 1             /* Non-compliant - begins with _ */
#undef _ BUILTIN_sqrt         /* Non-compliant - implementation may
                                 * use _BUILTIN_sqrt for other purposes,
                                 * e.g. generating a sqrt instruction */
#define defined               /* Non-compliant - reserved identifier */
#define errno my_errno        /* Non-compliant - library identifier */
#define isneg(x) ( (x) < 0 )  /* Compliant - rule doesn't include
                                 * future library directions   */
```

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 20.4` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.10

The Standard Library time and date functions shall not be used

Description

Rule Definition

The Standard Library time and date functions shall not be used.

Rationale

Using these functions can cause unspecified, undefined and implementation-defined behavior.

Polyspace Implementation

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.11

The standard header file <tgmath.h> shall not be used

Description

Rule Definition

The standard header file <tgmath.h> shall not be used.

Rationale

Using the facilities of this header file can cause undefined behavior.

Polyspace Implementation

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of Function in `tgmath.h`

```
#include <tgmath.h>

float f1,res;


void func(void) {
    res = sqrt(f1); /* Non-compliant */
}
```

In this example, the rule is violated when the `sqrt` macro defined in `tgmath.h` is used.

Correction — Use Appropriate Function in `math.h`

For this example, one possible correction is to use the function `sqrtf` defined in `math.h` for `float` arguments.

```
#include <math.h>

float f1, res;


void func(void) {
```

```
 res = sqrtf(f1);
}
```

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.12

The exception handling features of <fenv.h> should not be used

Description

Rule Definition

The exception handling features of <fenv.h> should not be used.

Rationale

In some cases, the values of the floating-point status flags are unspecified. Attempts to access them can cause undefined behavior.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of Features in <fenv.h>

```
#include <fenv.h>

void func(float x, float y) {
    float z;

    feclearexcept(FE_DIVBYZERO);              /* Non-compliant */
    z = x/y;

    if(fetestexcept (FE_DIVBYZERO))  {        /* Non-compliant */
    }
    else {
#pragma STDC FENV_ACCESS ON
        z=x*y;
        if(z>x) {
#pragma STDC FENV_ACCESS OFF
            if(fetestexcept (FE_OVERFLOW)) {  /* Non-compliant */
            }
        }
    }
}
```

In this example, the rule is violated when the identifiers `feclearexcept` and `fetestexcept`, and the macros `FE_DIVBYZERO` and `FE_OVERFLOW` are used.

Check Information

Group: Standard libraries
Category: Advisory
AGC Category: Advisory

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Rule 21.13

Any value passed to a function in <ctype.h> shall be representable as an unsigned char or be the value EOF

Description

Rule Definition

Any value passed to a function in <ctype.h> shall be representable as an unsigned char or be the value EOF.

Rationale

Functions in <ctype.h> have a well-defined behavior only for int arguments whose value is within the range of unsigned char or the negative value equivalent of EOF. The use of other values results in undefined behavior.

Polyspace Implementation

Polyspace considers that the negative value equivalent of EOF is -1 and does not raise a violation if you pass -1 as argument to a function in ctype.h.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Invalid Arguments for Functions from <ctype.h>

```
bool_t f (uint8_t a)
{
    return (         isdigit ((int32_t) a  )     /* Compliant     */
                &&  isalpha ((int32_t) 'b')     /* Compliant     */
                &&  islower (          EOF)     /* Compliant     */
                &&  isalpha (          256));   /* Non-compliant */
}
```

In this example, the rule is violated when 256, which is an neither an unsigned char or the value EOF, is passed as an input argument to the isalpha function.

Note The int casts in the above example are required to comply with Rule 10.3 on page 5-15.

Check Information

Group: Standard libraries
Category: Mandatory
AGC Category: Mandatory

See Also

`MISRA C:2012 Rule 10.3` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Rule 21.14

The Standard Library function `memcmp` shall not be used to compare null terminated strings

Description

Rule Definition

The Standard Library function `memcmp` shall not be used to compare null terminated strings.

Rationale

If `memcmp` is used to compare two strings and the length of either string is less than the number of bytes compared, the strings can appear different even when they are logically the same. The characters after the null terminator are compared even though they do not form part of the string.

For instance:

```
memcmp(string1, string2, sizeof(string1))
```

can compare bytes after the null terminator if `string1` is longer than `string2`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Using `memcmp` for String Comparison

```
extern char buffer1[ 12 ];
extern char buffer2[ 12 ];
void f1 ( void )
{
    ( void ) strcpy ( buffer1, "abc" );
    ( void ) strcpy ( buffer2, "abc" );
    if ( memcmp ( buffer1, buffer2, sizeof ( buffer1 ) ) != 0 )
    {
        /* Non-compliant */
    }
}
```

In this example, the comparison in the `if` statement is noncompliant. The strings stored in `buffer1` and `buffer2` can be reported different, but this difference comes from uninitialized characters after the null terminators.

Check Information

Group: Standard libraries
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 21.15` | `MISRA C:2012 Rule 21.16` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Rule 21.15

The pointer arguments to the Standard Library functions `memcpy`, `memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types

Description

Rule Definition

The pointer arguments to the Standard Library functions `memcpy`, `memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types.

Rationale

The functions

```
memcpy( arg1, arg2, num_bytes );
memmove( arg1, arg2, num_bytes );
memcmp( arg1, arg2, num_bytes );
```

perform a byte-by-byte copy, move or comparison between the memory locations that `arg1` and `arg2` point to. A byte-by-byte copy, move or comparison is meaningful only if `arg1` and `arg2` have compatible types.

Using pointers to different data types for `arg1` and `arg2` typically indicates a coding error.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Incompatible Argument Types for `memcpy`

```
void f ( uint8_t s1[ 8 ], uint16_t s2[ 8 ] )
{
    ( void ) memcpy ( s1, s2, 8 ); /* Non-compliant */
}
```

In this example, `s1` and `s2` are pointers to different data types. The `memcpy` statement copies eight bytes from one buffer to another.

Eight bytes represent the entire span of the buffer that `s1` points to, but only part of the buffer that `s2` points to. Therefore, the `memcpy` statement copies only part of `s2` to `s1`, which might be unintended.

Check Information

Group: Standard libraries
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 21.14` | `MISRA C:2012 Rule 21.16` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Rule 21.16

The pointer arguments to the Standard Library function `memcmp` shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type

Description

Rule Definition

The pointer arguments to the Standard Library function `memcmp` shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type.

Rationale

The Standard Library function

```
memcmp ( lhs, rhs, num );
```

performs a byte-by-byte comparison of the first `num` bytes of the two objects that `lhs` and `rhs` point to.

Do not use `memcmp` for a byte-by-byte comparison of the following.

Type	Rationale
Structures	If members of a structure have different data types, your compiler introduces additional padding for data alignment in memory. The content of these extra padding bytes is meaningless. If you perform a byte-by-byte comparison of structures with `memcmp`, you compare even the meaningless data stored in the padding. You might reach the false conclusion that two data structures are not equal, even if their corresponding members have the same value.
Objects with essentially floating type	The same floating point value can be stored using different representations. If you perform a byte-by-byte comparison of two variables with `memcmp`, you can reach the false conclusion that the variables are unequal even when they have the same value. The reason is that the values are stored using two different representations.
Essentially char arrays	Essentially char arrays are typically used to store strings. In strings, the content in bytes after the null terminator is meaningless. If you perform a byte-by-byte comparison of two strings with `memcmp`, you might reach the false conclusion that two strings are not equal, even if the bytes before the null terminator store the same value.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Using `memcmp` for Comparison of Structures, Unions, and *essentially char* Arrays

```
struct S;
bool_t f1 ( struct S *s1, struct S *s2 )
{
        return ( memcmp ( s1, s2, sizeof ( struct S ) ) != 0 ); /* Non-compliant */
}

union U
{
uint32_t range;
uint32_t height;
};
bool_t f2 ( union U *u1, union U *u2 )
{
        return ( memcmp ( u1, u2, sizeof ( union U ) ) != 0 ); /* Non-compliant */
}

const char a[ 6 ] = "task";
bool_t f3 ( const char b[ 6 ] )
{
        return ( memcmp ( a, b, 6 ) != 0 ); /* Non-compliant */
}
```

In this example:

- Structures `s1` and `s2` are compared in the `bool_t f1` function. The return value of this function might indicate that `s1` and `s2` are different due to padding. This comparison is noncompliant.
- Unions `u1` and `u2` are compared in the `bool_t f2` function. The return value of this function might indicate that `u1` and `u2` are the same due to unintentional comparison of `u1.range` and `u2.height`, or `u1.height` and `u2.range`. This comparison is noncompliant.
- Essentially char arrays `a` and `b` are compared in the `bool_t f3` function. The return value of this function might incorrectly indicate that the strings are different because the length of `a` (four) is less than the number of bytes compared (six). This comparison is noncompliant.

Check Information

Group: Standard libraries
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 21.14` | `MISRA C:2012 Rule 21.15` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Rule 21.17

Use of the string handling function from `<string.h>` shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters

Description

Rule Definition

Use of the string handling function from `<string.h>` shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters.

Rationale

Incorrect use of a string handling function might result in a read or write access beyond the bounds of the function arguments, resulting in undefined behavior.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Pointer Access Out of Bounds from `strcpy` Usage

```
char string[] = "Short";
void f1 ( const char *str )
{
        ( void ) strcpy ( string, "Too long to fit" );              /* Non-compliant */
        if ( strlen ( str ) < ( sizeof ( string ) - 1u ) )
        {
            ( void ) strcpy ( string, str );                         /* Compliant */
        }
}

size_t f2 ( void )
{
        char text[ 5 ] = "Token";
        return strlen ( text );                                     /* Non-compliant */
}
```

In this example:

- The first use of `strcpy` is noncompliant because it attempts to write beyond the end of its destination argument `string`.
- The second use of `strcpy` is compliant because it attempts to write to the destination argument `string` only if the source argument `str` fits.
- The use of `strlen` is noncompliant. `strlen` computes the length of a string up to the null terminator. The character array `text` has no null terminator.

Check Information

Group: Standard libraries
Category: Mandatory
AGC Category: Mandatory

See Also

`MISRA C:2012 Rule 21.18` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Rule 21.18

The `size_t` argument passed to any function in `<string.h>` shall have an appropriate value

Description

Rule Definition

The `size_t` argument passed to any function in `<string.h>` shall have an appropriate value.

Rationale

The value must be positive and not greater than the size of the smallest object passed by pointer to the function. For instance, suppose you use the `strncmp` function to compare two strings `lhs_string` and `rhs_string` as follows:

```
strncmp (lhs_string, rhs_string, num)
```

The third argument `num` must be positive and must not be greater than the size of `lhs_string` or `rhs_string`, whichever is smaller.

Otherwise, using the function can result in read or write access beyond the bounds of the function argument.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Incorrect `size_t` Argument for `memcmp`

```
char buf1[ 5 ] = "12345";
char buf2[ 10 ] = "1234567890";

void f ( void )
{
        if ( memcmp ( buf1, buf2, 5 ) == 0 )
        {
            /* Compliant */
        }
        if ( memcmp ( buf1, buf2, 6 ) == 0 )
        {
            /* Non-compliant */
        }
}
```

In this example, the first `if` statement is compliant. The `size_t` argument is five, which is same as the size of the smaller string, `buf1`.

By the same reasoning, the second `if` statement is noncompliant.

Check Information

Group: Standard libraries
Category: Mandatory
AGC Category: Mandatory

See Also

`MISRA C:2012 Rule 21.17` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Rule 21.19

The pointers returned by the Standard Library functions `localeconv`, `getenv`, `setlocale` or `strerror` shall only be used as if they have pointer to `const`-qualified type

Description

Rule Definition

The pointers returned by the Standard Library functions `localeconv`, `getenv`, `setlocale` or `strerror` shall only be used as if they have pointer to `const`-qualified type.

Rationale

The C99 Standard states that if the program modifies the structure pointed to by the value returned by `localeconv`, or the strings returned by `getenv`, `setlocale` or `strerro`, undefined behavior occurs. Treating the pointers returned by the various functions as if they were `const`-qualified allows an analysis tool to detect any attempt to modify an object through one of the pointers. Assigning the return values of the functions to `const`-qualified pointers results in the compiler issuing a diagnostic if an attempt is made to modify an object.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Returning Pointers from`setlocale` and `localeconv`

```
void f1 ( void )
{
        char *s1 = setlocale ( LC_ALL, 0 ); /* Non-compliant */
        struct lconv *conv = localeconv (); /* Non-compliant */
        s1[ 1 ] = 'A'; /* Undefined behavior */
        conv->decimal_point = "^"; /* Undefined behavior */
}

void f2 ( void )
{
        char str[128];
        (void) strcpy (str, setlocale ( LC_ALL,0 ) ); /* Compliant */
        const struct lconv *conv = localeconv ();     /* Compliant */
        conv->decimanl_point = "^"                    /* Constraint violation */
}

void f3 ( void )
{
const struct lconv *conv = localeconv (); /* Compliant */
conv->grouping[ 2 ] = 'x';                /* Non-compliant */
}
```

In the above example:

- The usage of `setlocale` and `localeconv` in the function `f1` are non-compliant as the returned pointers are assigned to non-`const`—qualified pointers.

  Note The usage of `setlocale` and `localeconv` above are not constraint violations and will therefore not be reported by a compiler. However, an analysis tool will be able to report a violation.

- The usage of `setlocale` in the function `f2` is compliant as `strcpy` takes a `const char *` as its second parameter. The usage of `localeconv` in the function `f2` is compliant as the returned pointers are assigned to a `const`-qualified pointer. Any attempt to modify an object through a pointer will be reported by a compiler or analysis tool as this is a constraint violation.
- The usage of a `const`-qualified pointer in the function `f3` gives compile time protection of the value returned by `localeconv` but the same is not true for the strings it references. Modification of these strings can be detected by an analysis tool.

Check Information

Group: Standard libraries
Category: Mandatory
AGC Category: Mandatory

See Also

`MISRA C:2012 Rule 7.4` | `MISRA C:2012 Rule 11.8` | `MISRA C:2012 Rule 21.8` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Rule 21.2

A reserved identifier or macro name shall not be declared

Description

Rule Definition

A reserved identifier or macro name shall not be declared.

Rationale

The Standard allows implementations to treat reserved identifiers specially. If you reuse reserved identifiers, you can cause undefined behavior.

Polyspace Implementation

- If you define a macro name that corresponds to a standard library macro, object, or function, rule 21.1 is violated.
- The rule considers tentative definitions as definitions.

Additional Message in Report

Identifier 'XX' shall not be reused.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.20

The pointer returned by the Standard Library functions `asctime`, `ctime`, `gmtime`, `localtime`, `localeconv`, `getenv`, `setlocale` or `strerror` shall not be used following a subsequent call to the same function

Description

Rule Definition

The pointer returned by the Standard Library functions `asctime`, `ctime`, `gmtime`, `localtime`, `localeconv`, `getenv`, `setlocale` or `strerror` shall not be used following a subsequent call to the same function.

Rationale

The preceding functions return a pointer to an object within the Standard Library. Implementation for this object can use a static buffer that can be modified by a second call to the same function. Therefore the value accessed through a pointer before a subsequent call to the same function can change unexpectedly.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of Return Value from `getenv` After Another Call to `getenv`

```
void f1( void )
{
        const char *res1;
        const char *res2;
        char copy[ 128 ];
        res1 = setlocale ( LC_ALL, 0 );
        ( void ) strcpy ( copy, res1 );
        res2 = setlocale ( LC_MONETARY, "French" );
        printf ( "%s\n", res1 ); /* Non-compliant */
        printf ( "%s\n", copy ); /* Compliant */
        printf ( "%s\n", res2 ); /* Compliant */
}
```

In this example:

- The first `printf` statement is non-compliant because the pointer returned by `setlocale` is used following a subsequent call to it when `res2` is assigned.
- The second `printf` statement is compliant because the copy operation performed by `strcpy` is made before a subsequent call to `setlocale` function is made.
- The third `printf` statement is compliant because there is no subsequent call to the `setlocale` function is made before use.

Check Information

Group: Standard libraries
Category: Mandatory
AGC Category: Mandatory

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Rule 21.3

The memory allocation and deallocation functions of <stdlib.h> shall not be used

Description

Rule Definition

The memory allocation and deallocation functions of <stdlib.h> shall not be used.

Rationale

Using memory allocation and deallocation routines can cause undefined behavior. For instance:

- You free memory that you had not allocated dynamically.
- You use a pointer that points to a freed memory location.

Polyspace Implementation

If you use names of dynamic heap memory allocation functions for macros, and you expand the macros in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro <name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `malloc`, `calloc`, `realloc` and `free`

```
#include <stdlib.h>

static int foo(void);

typedef struct struct_1 {
    int a;
    char c;
} S_1;

static int foo(void) {

    S_1 * ad_1;
    int  * ad_2;
    int  * ad_3;

    ad_1 = (S_1*)calloc(100U, sizeof(S_1));        /* Non-compliant */
    ad_2 = malloc(100U * sizeof(int));             /* Non-compliant */
    ad_3 = realloc(ad_3, 60U * sizeof(long));      /* Non-compliant */
```

```
    free(ad_1);                                     /* Non-compliant */
    free(ad_2);                                     /* Non-compliant */
    free(ad_3);                                     /* Non-compliant */

    return 1;
}
```

In this example, the rule is violated when the functions `malloc`, `calloc`, `realloc` and `free` are used.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 18.7` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.4

The standard header file <setjmp.h> shall not be used

Description

Rule Definition

The standard header file <setjmp.h> shall not be used.

Rationale

Using `setjmp` and `longjmp`, you can bypass normal function call mechanisms and cause undefined behavior.

Polyspace Implementation

If the `longjmp` function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.5

The standard header file <signal.h> shall not be used

Description

Rule Definition

The standard header file <signal.h> shall not be used.

Rationale

Using signal handling functions can cause implementation-defined and undefined behavior.

Polyspace Implementation

If the signal function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.6

The Standard Library input/output functions shall not be used

Description

Rule Definition

The Standard Library input/output functions shall not be used.

Rationale

This rule applies to the functions that are provided by `<stdio.h>` and in C99, their character-wide equivalents provided by `<wchar.h>`. Using these functions can cause unspecified, undefined and implementation-defined behavior.

Polyspace Implementation

If the Standard Library function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.7

The `atof`, `atoi`, `atol`, and `atoll` functions of `<stdlib.h>` shall not be used

Description

Rule Definition

The atof, atoi, atol, and atoll functions of <stdlib.h> shall not be used.

Rationale

When a string cannot be converted, the behavior of these functions can be undefined.

Polyspace Implementation

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.8

The library functions abort, exit, getenv and system of <stdlib.h> shall not be used

Description

Rule Definition

The library functions abort, exit, getenv and system of <stdlib.h> shall not be used.

Rationale

Using these functions can cause undefined and implementation-defined behaviors.

Polyspace Implementation

In case the abort, exit, getenv, and system functions are actually macros, and the macros are expanded in the code, this rule is detected as violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.9

The library functions bsearch and qsort of <stdlib.h> shall not be used

Description

Rule Definition

The library functions bsearch and qsort of <stdlib.h> shall not be used.

Rationale

The comparison function in these library functions can behave inconsistently when the elements being compared are equal. Also, the implementation of `qsort` can be recursive and place unknown demands on the call stack.

Polyspace Implementation

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 22.1

All resources obtained dynamically by means of Standard Library functions shall be explicitly released

Description

Rule Definition

All resources obtained dynamically by means of Standard Library functions shall be explicitly released.

Rationale

Resources are something that you must return to the system once you have used them. Examples include dynamically allocated memory and file descriptors.

If you do not release resources explicitly as soon as possible, then a failure can occur due to exhaustion of resources.

Polyspace Implementation

You can check for this rule with a Bug Finder analysis only.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Dynamic Memory

```
#include<stdlib.h>


void performOperation(int);

int func1(int num) {
    int *arr1 = (int*) malloc(num * sizeof(int));

    return 0;
}           /* Non-compliant - memory allocated to arr1 is not released */

int func2(int num) {
    int *arr2 = (int*) malloc(num * sizeof(int));

    free(arr2);
    return 0;
}          /* Compliant - memory allocated to arr2 is released */
```

In this example, the rule is violated when memory dynamically allocated using the `malloc` function is not freed using the `free` function before the end of scope.

File Pointers

```
#include <stdio.h>
void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );

    fp1 = fopen ( "data2.txt", "w" );              /* Non-compliant */
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}

void func2( void ) {
    FILE *fp2;
    fp2 = fopen ( "data1.txt", "w" );
    fprintf ( fp2, "*" );
    fclose(fp2);

    fp2 = fopen ( "data2.txt", "w" );              /* Compliant */
    fprintf ( fp2, "!" );
    fclose ( fp2 );
}
```

In this example, the file pointer `fp1` is pointing to a file `data1.txt`. Before `fp1` is explicitly dissociated from the file stream of `data1.txt`, it is used to access another file `data2.txt`. Therefore, the rule 22.1 is violated.

The rule is not violated in `func2` because file `data1.txt` is closed and the file pointer `fp2` is explicitly dissociated from `data1.txt` before it is reused.

Check Information

Group: Resources
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)|MISRA C:2012 Dir 4.13|MISRA C:2012 Rule 21.3|MISRA C:2012 Rule 21.6|Resource leak`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Rule 22.10

The value of `errno` shall only be tested when the last function to be called was an `errno`-setting function

Description

Rule Definition

The value of `errno` shall only be tested when the last function to be called was an `errno`-setting function.

Rationale

Besides the `errno`-setting functions, the Standard does not enforce that other functions set `errno` on errors. Whether these functions set `errno` or not is implementation-dependent.

To detect errors, if you check `errno` alone, the validity of this check also becomes implementation-dependent. On implementations that do not require `errno` setting, even if you check `errno` alone, you can overlook error conditions.

For a list of `errno`-setting functions, see `MISRA C:2012 Rule 22.8`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Incorrect Test of `errno`

```
void f ( void )
{
    float64_t f64;
    errno = 0;
    f64 = atof ( "A.12" );
    if ( 0 == errno ) /* Non-compliant */
    {
    }
    errno = 0;
    f64 = strtod ( "A.12", NULL );
    if ( 0 == errno ) /* Compliant */
    {
    }
}
```

In this example:

- The first `if` statement is noncompliant because `atof` may or may not set `errno` when an error is detected. `f64` may not have a valid value within this `if` statement.
- The second `if` statement is compliant because `strtod` is an *`errno`-setting function*. `f64` will have a valid value within this `if` statement.

Check Information

Group: Resources
Category: Required
AGC Category: Required

See Also

MISRA C:2012 Rule 22.8 | MISRA C:2012 Rule 22.9 | Check MISRA C:2012 (-misra3)

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Rule 22.2

A block of memory shall only be freed if it was allocated by means of a Standard Library function

Description

Rule Definition

A block of memory shall only be freed if it was allocated by means of a Standard Library function.

Rationale

The Standard Library functions that allocate memory are `malloc`, `calloc` and `realloc`.

You free a block of memory when you pass its address to the `free` or `realloc` function. The following causes undefined behavior:

- You free a block of memory that you did not allocate.
- You free a block of memory that have already freed before.

Polyspace Implementation

You can check for this rule with a Bug Finder analysis only.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Memory Not Allocated Is Freed

```
#include <stdlib.h>

void func1(void) {
    int x=0;
    int *ptr=&x;

    free(ptr);
   /* Non-compliant: ptr is not dynamically allocated */
}
```

In this example, the rule is violated because the `free` function operates on a pointer that does not point to dynamically allocated memory.

Memory Freed Twice

```
#include <stdlib.h>

void func(int arrSize) {
    int *ptr = (int*) malloc(arrSize* sizeof(int));

    free(ptr);   /* Block of memory freed once */
```

```
    free(ptr);   /* Non-compliant - Block of memory freed twice */
}
```

In this example, the rule is violated when the `free` function operates on `ptr` twice without a reallocation in between.

Check Information

Group: Resources
Category: Mandatory
AGC Category: Mandatory

See Also

`Check MISRA C:2012 (-misra3)|Deallocation of previously deallocated pointer|Invalid free of pointer|MISRA C:2012 Dir 4.13|MISRA C:2012 Rule 21.3`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Rule 22.3

The same file shall not be open for read and write access at the same time on different streams

Description

Rule Definition

The same file shall not be open for read and write access at the same time on different streams.

Rationale

If a file is both written and read via different streams, the behavior can be undefined.

Polyspace Implementation

You can check for this rule with a Bug Finder analysis only.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Opening File That Is Open in Another Stream

```
#include <stdio.h>

void func(void) {
    FILE *fw = fopen("tmp.txt", "r+");
    FILE *fr = fopen("tmp.txt", "r");   /* Non-compliant: File open in stream fw*/
}
```

In this example, the rule is violated when the same file `tmp.txt` is opened in two streams. The `FILE` pointers `fw` and `fr` point to two different streams here.

Check Information

Group: Resources
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)|MISRA C:2012 Rule 21.6|Resource leak`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Rule 22.4

There shall be no attempt to write to a stream which has been opened as read-only

Description

Rule Definition

There shall be no attempt to write to a stream which has been opened as read-only.

Rationale

The Standard does not specify the behavior if an attempt is made to write to a read-only stream.

Polyspace Implementation

You can check for this rule with a Bug Finder analysis only.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Writing to File Opened as Read-Only

```
#include <stdio.h>

void func1(void) {
    FILE *fp1 = fopen("tmp.txt", "r");
    (void) fprintf(fp1, "Some text"); /* Non-compliant: Read-only stream */
    (void) fclose(fp1);
}

void func2(void) {
    FILE *fp2 = fopen("tmp.txt", "r+");
    (void) fprintf(fp2, "Some text"); /* Compliant */
    (void) fclose(fp2);
}
```

In this example, the file stream associated with `fp1` is opened as read-only. The rule is violated when the stream is written.

Check Information

Group: Resources
Category: Mandatory
AGC Category: Mandatory

See Also

`Check MISRA C:2012 (-misra3)|MISRA C:2012 Rule 21.6|Writing to read-only resource`

Topics
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Rule 22.5

A pointer to a FILE object shall not be dereferenced

Description

Rule Definition

A pointer to a FILE object shall not be dereferenced.

Rationale

The Standard states that the address of a FILE object used to control a stream can be significant. Copying that object might not give the same behavior. This rule ensures that you cannot perform such a copy.

Directly manipulating a FILE object might be incompatible with its use as a stream designator.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

FILE* Pointer Dereferenced

```
#include <stdio.h>

void func(void) {
    FILE *pf1;
    FILE *pf2;
    FILE f3;

    pf2 = pf1;        /* Compliant */
    f3 = *pf2;        /* Non-compliant */
    pf2->_flags=0;    /* Non-compliant */
 }
```

In this example, the rule is violated when the FILE* pointer pf2 is dereferenced.

Check Information

Group: Resources
Category: Mandatory
AGC Category: Mandatory

See Also

Check MISRA C:2012 (-misra3)|MISRA C:2012 Rule 21.6

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Rule 22.6

The value of a pointer to a FILE shall not be used after the associated stream has been closed

Description

Rule Definition

The value of a pointer to a FILE shall not be used after the associated stream has been closed.

Rationale

The Standard states that the value of a FILE* pointer is indeterminate after you close the stream associated with it.

Polyspace Implementation

You can check for this rule with a Bug Finder analysis only.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of FILE Pointer After Closing Stream

```
#include <stdio.h>

void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fclose(fp);
        fprintf(fp,"text");
    }
}
```

In this example, the stream associated with the FILE* pointer fp is closed with the fclose function. The rule is violated FILE* pointer fp is used before the stream is re-opened.

Check Information

Group: Resources
Category: Mandatory
AGC Category: Mandatory

See Also

Check MISRA C:2012 (-misra3)|MISRA C:2012 Dir 4.13|MISRA C:2012 Rule 21.6|Use of previously closed resource

Topics
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Rule 22.7

The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF

Description

Rule Definition

The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF.

Rationale

The EOF value may become indistinguishable from a valid character code if the value returned is converted to another type. In such cases, testing the converted value against EOF will not reliably identify if the end of the file has been reached or if an error has occurred.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Possibly Misleading Results from Comparison with EOF

```
void f1 ( void )
{
    char ch;
    ch = ( char ) getchar ();
    if ( EOF != ( int32_t ) ch )  /* Non-compliant */
    {
    }
}

void f2 ( void )
{
    char ch;
    ch = ( char ) getchar ();
    if ( !feof ( stdin ) )        /* Compliant */
    {
    }
}

void f3 ( void )
{
    int32_t i_ch;
    i_ch = getchar ();
    if ( EOF != i_ch )            /* Compliant */
     {
        char ch;
        ch = ( char ) i_ch;
    }
}
```

In this example:

- The test in the `f1` function is non-compliant. It will not be reliable as the return value is cast to a narrower type before checking for `EOF`.
- The test in the `f2` function is compliant. It shows how *`feof()`* can be used to check for `EOF` when the return value from *`getchar()`* has been subjected to type conversion.
- The test in the `f3` function is compliant. It is reliable as the unconverted return value is used when checking for `EOF`.

Check Information

Group: Resources
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Rule 22.8

The value of `errno` shall be set to zero prior to a call to an `errno`-setting-function

Description

Rule Definition

The value of `errno` shall be set to zero prior to a call to an `errno`-setting-function.

Rationale

If an error occurs during a call to an `errno`-setting-function, the function writes a nonzero value to `errno`. Otherwise, `errno` is not modified.

If you do not explicitly set `errno` to zero before a function call, it can contain values from a previous call. Checking `errno` for nonzero values after the function call can give the false impression that an error occurred.

`Errno`-setting functions include:

- `ftell`, `fgetpos`, `fgetwc` and related functions.
- `strtoimax`, `strtol` and related functions.

  The wide-character equivalents such as `wcstoimax` and `wcstol` are also covered.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

`errno` Not Reset Before Use

```
#include <stdlib.h>
#include <errno.h>

double val = 0.0;

void f ( void )
{
    val = strtod("1.0",NULL); /* Non-compliant */
    if ( 0 == errno ) /* Check errno for nonzero values */
    {
        val = strtod("1.0",NULL); /* Compliant - case 1*/
        if ( 0 == errno ) /* Check errno for nonzero values */
        {
        }
    }
    else
    {
        errno = 0;
        val = strtod("1.0",NULL); /* Compliant - case 2*/
```

```
        if ( 0 == errno ) /* Check errno for nonzero values */
        {
        }
    }
}
```

In this example, the rule is violated when `strtod` is called but `errno` is not reset prior to the call.

The rule is not violated in the following cases:

- Case 1: `errno` is compared against zero and then `strtod` is called in the `if( 0 == errno )` branch.
- Case 2: `errno` is explicitly set to zero and then `strtod` is called.

Check Information

Group: Resources
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 22.9` | `MISRA C:2012 Rule 22.10` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Rule 22.9

The value of `errno` shall be tested against zero after calling an `errno`-setting function

Description

Rule Definition

The value of `errno` shall be tested against zero after calling an `errno`-setting function.

Rationale

If an error occurs during a call to an `errno`-setting-function, the function writes a nonzero value to `errno`. Otherwise, `errno` is not modified.

When `errno` is nonzero, the function return value is not likely to be correct. Before using this return value, you must test `errno` for nonzero values.

`Errno`-setting functions include:

- `ftell`, `fgetpos`, `fgetwc` and related functions.
- `strtoimax`, `strtol` and related functions.

  The wide-character equivalents such as `wcstoimax` and `wcstol` are also covered.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

`errno` Not Tested After Function Call

```
#include <stdlib.h>
#include <errno.h>

void func(void);
double val = 0.0;

void f1 ( void )
{
  errno = 0;
  val = strtod ( "1.0", NULL ); /* Non-compliant */
  func ();

  if ( 0 != errno )
    {
    }

  errno = 0;
  val = strtod ( "1.0", NULL ); /* Compliant */
  if ( 0 == errno )
    {
      func();
    }
}
```

In this example, the rule is violated when `errno` is not checked immediately after the first call to `strtod`. Instead, a second function `func` is called. `func` might use the value in the global variable `val`. The value can be incorrect if an error has occurred during the call to `strtod`.

The rule is not violated when `errno` is checked before operations that potentially use the return value of `strtod`.

Check Information

Group: Resources
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 22.8` | `MISRA C:2012 Rule 22.10` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Rule 3.1

The character sequences /* and // shall not be used within a comment

Description

Rule Definition

The character sequences / and // shall not be used within a comment.*

Rationale

These character sequences are not allowed in code comments because:

- If your code contains a /* or a // in a /* */ comment, it typically means that you have inadvertently commented out code.
- If your code contains a /* in a // comment, it typically means that you have inadvertently uncommented a /* */ comment.

Polyspace Implementation

You cannot annotate this rule in the source code.

For information on annotations, see “Annotate Code and Hide Known or Acceptable Results”.

Additional Message in Report

The character sequence /* shall not appear within a comment.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

/* Used in // Comments

```
int x;
int y;
int z;

void non_compliant_comments ( void )
{
    x = y //      /* Non-compliant
        + z
        //  */
        ;
    z++;    //    Compliant with exception: // permitted within a // comment
}

void compliant_comments ( void )
{
    x = y /*      Compliant
```

```
      + z
      */
        ;
    z++;    //    Compliant with exception: // is permitted within a // comment
}
```

In this example, in the `non_compliant_comments` function, the `/*` character occurs in what appears to be a `//` comment, violating the rule. Because of the comment structure, the operation that takes place is `x = y + z;`. However, without the two `//`-s, an entirely different operation `x=y;` takes place. It is not clear which operation is intended.

Use a comment format that makes your intention clear. For instance, in the `compliant_comments` function, it is clear that the operation `x=y;` is intended.

Check Information

Group: Comments
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 3.2

Line-splicing shall not be used in `//` comments

Description

Rule Definition

Line-splicing shall not be used in // comments.

Rationale

Line-splicing occurs when the `\` character is immediately followed by a new-line character. Line splicing is used for statements that span multiple lines.

If you use line-splicing in a `//` comment, the following line can become part of the comment. In most cases, the `\` is spurious and can cause unintentional commenting out of code.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Line Splicing in `//` Comment

```
#include <stdbool.h>

extern _Bool b;

void func ( void )
{
    unsigned short x = 0;   // Non-compliant - Line-splicing \
    if ( b )
    {
        ++b;
    }
}
```

Because of line-splicing, the statement `if ( b )` is a part of the previous `//` comment. Therefore, the statement `b++` always executes, making the `if` block redundant.

Check Information

Group: Comments
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”

“Polyspace MISRA C:2012 Checkers”

Introduced in R2014b

MISRA C:2012 Rule 4.1

Octal and hexadecimal escape sequences shall be terminated

Description

Rule Definition

Octal and hexadecimal escape sequences shall be terminated.

Rationale

There is potential for confusion if an octal or hexadecimal escape sequence is followed by other characters. For example, the character constant `'\x1f'` consists of a single character, whereas the character constant `'\x1g'` consists of the two characters `'\x1'` and `'g'`. The manner in which multi-character constants are represented as integers is implementation-defined.

If every octal or hexadecimal escape sequence in a character constant or string literal is terminated, you reduce potential confusion.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Compliant and Noncompliant Escape Sequences

```
const char *s1 = "\x41g";     /* Non-compliant */
const char *s2 = "\x41" "g";  /* Compliant - Terminated by end of literal */
const char *s3 = "\x41\x67";  /* Compliant - Terminated by another escape sequence*/

int c1 = '\141t';             /* Non-compliant */
int c2 = '\141\t';            /* Compliant - Terminated by another escape sequence*/
```

In this example, the rule is violated when an escape sequence is not terminated with the end of string literal or another escape sequence.

Check Information

Group: Character Sets and Lexical Conventions
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 4.2

Trigraphs should not be used

Description

Rule Definition

Trigraphs should not be used.

Rationale

You denote trigraphs with two question marks followed by a specific third character (for instance,`'??-'` represents a `'~'` (tilde) character and `'??)'` represents a `']'`). These trigraphs can cause accidental confusion with other uses of two question marks.

Note Digraphs (`<: :>`, `<% %>`, `%:`, `%:%:`) are permitted because they are tokens.

Polyspace Implementation

The Polyspace analysis converts trigraphs to the equivalent character for the defect analysis. However, Polyspace also raises a MISRA violation.

The standard requires that trigraphs must be transformed *before* comments are removed during preprocessing. Therefore, Polyspace raises a violation of this rule even if a trigraph appears in code comments.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Character Sets and Lexical Conventions
Category: Advisory
AGC Category: Advisory

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.1

External identifiers shall be distinct

Description

Rule Definition

External identifiers shall be distinct.

Rationale

External identifiers are ones declared with global scope or storage class `extern`.

If the difference between two names occurs far later in the names, they can be easily mistaken for each other. The readability of the code is reduced.

Polyspace Implementation

Polyspace considers two names as distinct if there is a difference between their first 31 characters. For C90, the difference must occur between the first 6 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`.

Additional Message in Report

External `%s %s` conflicts with the external identifier XX in file YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

C90: First Six Characters of Identifiers Not Unique

```
int engine_temperature_raw;
int engine_temperature_scaled;   /* Non-compliant */
int engin2_temperature;          /* Compliant */
```

In this example, the identifier `engine_temperature_scaled` has the same first six characters as a previous identifier, `engine_temperature_raw`.

C99: First 31 Characters of Identifiers Not Unique

```
int engine_exhaust_gas_temperature_raw;
int engine_exhaust_gas_temperature_scaled; /* Non-compliant */

int eng_exhaust_gas_temp_raw;
int eng_exhaust_gas_temp_scaled;           /* Compliant */
```

In this example, the identifier `engine_exhaust_gas_temperature_scaled` has the same first 31 characters as a previous identifier, `engine_exhaust_gas_temperature_raw`.

C90: First Six Characters Identifiers in Different Translation Units Differ in Case Alone

```
/* file1.c */
int abc = 0;

/* file2.c */
int ABC = 0; /* Non-compliant */
```

In this example, the implementation supports 6 significant case-insensitive characters in *external identifiers*. The identifiers in the two translation are different but are not distinct in their significant characters.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 5.2` | `MISRA C:2012 Rule 5.4` | `MISRA C:2012 Rule 5.5` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.2

Identifiers declared in the same scope and name space shall be distinct

Description

Rule Definition

Identifiers declared in the same scope and name space shall be distinct.

Rationale

If the difference between two names occurs far later in the names, they can be easily mistaken for each other. The readability of the code is reduced.

Polyspace Implementation

Polyspace considers two names as distinct if there is a difference between their first 63 characters. In C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`.

Additional Message in Report

Identifier XX has same significant characters as identifier YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

C90: First 31 Characters of Identifiers Not Unique

```
extern int engine_exhaust_gas_temperature_raw;
static int engine_exhaust_gas_temperature_scaled;      /* Non-compliant */

extern double engine_exhaust_gas_temperature_raw;
static double engine_exhaust_gas_temperature2_scaled;  /* Compliant */

void func ( void )
{
  /* Not in the same scope */
  int engine_exhaust_gas_temperature_local;            /* Compliant */
}
```

In this example, the identifier `engine_exhaust_gas_temperature_scaled` has the same 31 characters as a previous identifier, `engine_exhaust_gas_temperature_raw`.

The rule does not apply if the two identifiers have the same 31 characters but have different scopes. For instance, `engine_exhaust_gas_temperature_local` has the same 31 characters as `engine_exhaust_gas_temperature_raw` but different scope.

C99: First 63 Characters of Identifiers Not Unique

```
extern int engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_raw;
static int engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_scale;
    /* Non-compliant */

extern int engine_gas_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx__raw;
static int engine_gas_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx__scale;
    /* Compliant */

void func ( void )
{
/* Not in the same scope */
    int engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_local;
          /* Compliant */
}
```

In this example, the identifier `engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_scale` has the same 63 characters as a previous identifier, `engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_raw`.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 5.1` | `MISRA C:2012 Rule 5.3` | `MISRA C:2012 Rule 5.4` | `MISRA C:2012 Rule 5.5` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.3

An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

Description

Rule Definition

An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.

Rationale

If two identifiers have the same name but different scope, the identifier in the inner scope hides the identifier in the outer scope. All uses of the identifier name refers to the identifier in the inner scope. This behavior forces the developer to keep track of the scope and reduces code readability.

Polyspace Implementation

Polyspace considers two names as distinct if there is a difference between their first 63 characters. In C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`.

If the identifier that is hidden is declared in a Standard Library header and you do not provide the header for the analysis, the issue is not shown. To see potential conflicts with identifiers declared in a Standard Library header, provide your compiler implementation of the headers for the Polyspace analysis. See “Provide Standard Library Headers for Polyspace Analysis”.

Additional Message in Report

Variable XX hides variable XX (FILE line LINE column COLUMN).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Local Variable Hidden by Another Local Variable in Inner Block

```
typedef signed short int16_t;

void func( void )
{
    int16_t i;
    {
        int16_t i;                /* Non-compliant */
        i = 3;
    }
}
```

In this example, the identifier `i` defined in the inner block in `func` hides the identifier `i` with function scope.

It is not immediately clear to a reader which `i` is referred to in the statement `i=3`.

Global Variable Hidden by Function Parameter

```
typedef signed short int16_t;

struct astruct
{
    int16_t m;
};

extern void g ( struct astruct *p );
int16_t xyz = 0;

void func ( struct astruct xyz )  /* Non-compliant */
{
    g ( &xyz );
}
```

In this example, the parameter `xyz` of function `func` hides the global variable `xyz`.

It is not immediately clear to a reader which `xyz` is referred to in the statement `g (&xyz )`.

Check Information

Group: Identifiers
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 5.2` | `MISRA C:2012 Rule 5.8` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.4

Macro identifiers shall be distinct

Description

Rule Definition

Macro identifiers shall be distinct.

Rationale

The names of macro identifiers must be distinct from both other macro identifiers and their parameters.

Polyspace Implementation

Polyspace considers two names as distinct if there is a difference between their first 63 characters. In C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`.

Additional Message in Report

- Macro identifiers shall be distinct. Macro XX has same significant characters as macro YY.
- Macro identifiers shall be distinct. Macro parameter XX has same significant characters as macro parameter YY in macro ZZ.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

C90: First 31 Characters of Macro Names Not Unique

```
#define engine_exhaust_gas_temperature_raw egt_r
#define engine_exhaust_gas_temperature_scaled egt_s   /* Non-compliant */

#define engine_exhaust_gas_temp_raw egt_r
#define engine_exhaust_gas_temp_scaled egt_s          /* Compliant */
```

In this example, the macro `engine_exhaust_gas_temperature_scaled egt_s` has the same first 31 characters as a previous macro `engine_exhaust_gas_temperature_scaled`.

C99: First 63 Characters of Macro Names Not Unique

```
#define engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_raw egt_r
#define engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_raw_scaled egt_s
    /* Non-compliant */

/* 63 significant case-sensitive characters in macro identifiers */
#define new_engine_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_raw egt_r
```

```
#define new_engine_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_scaled egt_s
    /* Compliant */
```

In this example, the macro `engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx___gaz_scaled` has the same first 63 characters as a previous macro `engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx___raw`.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 5.1` | `MISRA C:2012 Rule 5.2` | `MISRA C:2012 Rule 5.5` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.5

Identifiers shall be distinct from macro names

Description

Rule Definition

Identifiers shall be distinct from macro names.

Rationale

The rule requires that macro names that exist only prior to processing must be different from identifier names that also exist after preprocessing. Keeping macro names and identifiers distinct help avoid confusion.

Polyspace Implementation

Polyspace considers two names as distinct if there is a difference between their first 63 characters. In C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`.

Additional Message in Report

Identifier XX has same significant characters as macro YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Macro Names Same as Identifier Names

```
#define Sum_1(x, y) ( ( x ) + ( y ) )
short Sum_1;                    /* Non-compliant */

#define Sum_2(x, y) ( ( x ) + ( y ) )
short x = Sum_2 ( 1, 2 );       /* Compliant */
```

In this example, `Sum_1` is both the name of an identifier and a macro. `Sum_2` is used only as a macro.

C90: First 31 Characters of Macro Name Same as Identifier Name

```
#define       low_pressure_turbine_temperature_1 lp_tb_temp_1
static int low_pressure_turbine_temperature_2;      /* Non-compliant  */
```

In this example, the identifier `low_pressure_turbine_temperature_2` has the same first 31 characters as a previous macro `low_pressure_turbine_temperature_1`.

Check Information

Group: Identifiers

Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 5.1` | `MISRA C:2012 Rule 5.2` | `MISRA C:2012 Rule 5.4` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.6

A typedef name shall be a unique identifier

Description

Rule Definition

A typedef name shall be a unique identifier.

Rationale

Reusing a `typedef` name as another `typedef` or as the name of a function, object or `enum` constant can cause developer confusion.

Additional Message in Report

XX conflicts with the typedef name YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

`typedef` Names Reused

```
void func ( void ){
  {
    typedef unsigned char u8_t;
  }
  {
    typedef unsigned char u8_t; /* Non-compliant */
  }
}

typedef float mass;
void func1 ( void ){
  float mass = 0.0f;            /* Non-compliant */
}
```

In this example, the `typedef` name `u8_t` is used twice. The `typedef` name `mass` is also used as an identifier name.

`typedef` Name Same as Structure Name

```
typedef struct list{          /* Compliant - exception */
  struct list *next;
  unsigned short element;
} list;

typedef struct{
  struct chain{               /* Non-compliant */
    struct chain *list2;
```

```
    unsigned short element;
  } s1;
  unsigned short length;
} chain;
```

In this example, the `typedef` name `list` is the same as the original name of the `struct` type. The rule allows this exceptional case.

However, the `typedef` name `chain` is not the same as the original name of the `struct` type. The name `chain` is associated with a different `struct` type. Therefore, it clashes with the `typedef` name.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 5.7 | Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.7

A tag name shall be a unique identifier

Description

Rule Definition

A tag name shall be a unique identifier.

Rationale

Reusing a tag name can cause developer confusion.

Additional Message in Report

XX conflicts with the tag name YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 5.6` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.8

Identifiers that define objects or functions with external linkage shall be unique

Description

Rule Definition

Identifiers that define objects or functions with external linkage shall be unique.

Rationale

External identifiers are those declared with global scope or with storage class `extern`. Reusing an external identifier name can cause developer confusion.

Identifiers defined within a function have smaller scope. Even if names of such identifiers are not unique, they are not likely to cause confusion.

Additional Message in Report

- Object XX conflicts with the object name YY.
- Function XX conflicts with the function name YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 5.3|Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.9

Identifiers that define objects or functions with internal linkage should be unique

Description

Rule Definition

Identifiers that define objects or functions with internal linkage should be unique.

Polyspace Implementation

This rule checker assumes that rule 5.8 is not violated.

Additional Message in Report

- Object XX conflicts with the object name YY.
- Function XX conflicts with the function name YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Identifiers
Category: Advisory
AGC Category: Readability

See Also

`MISRA C:2012 Rule 8.10` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 6.1

Bit-fields shall only be declared with an appropriate type

Description

Rule Definition

Bit-fields shall only be declared with an appropriate type.

Rationale

Using `int` for a bit-field type is implementation-defined because bit-fields of type `int` can be either `signed` or `unsigned`.

The use of `enum`, `short char`, or any other type of bit-field is not permitted in C90 because the behavior is undefined.

In C99, the implementation can potentially define other integer types that are permitted in bit-field declarations.

Polyspace Implementation

The checker flags data types for bit-fields other than these allowed types:

- C90: `signed int` or `unsigned int` (or `typedef`-s that resolve to these types)
- C99: `signed int`, `unsigned int` or `_Bool` (or `typedef`-s that resolve to these types)

The results depend on the version of the C standard used in the analysis. See `C standard version (-c-version)`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Types
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 6.2

Single-bit named bit fields shall not be of a signed type

Description

Rule Definition

Single-bit named bit fields shall not be of a signed type.

Rationale

According to the C99 Standard Section 6.2.6.2, a single-bit signed bit-field has one sign bit and no value bits. In any representation of integers, zero value bits cannot specify a meaningful value.

A single-bit signed bit-field is therefore unlikely to behave in a useful way. Its presence is likely to indicate programmer confusion.

Although the C90 Standard does not provide much detail regarding the representation of types, the same single-bit bit-field considerations apply.

Polyspace Implementation

This rule does not apply to unnamed bit fields because their values cannot be accessed.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Types
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 7.1

Octal constants shall not be used

Description

Rule Definition

Octal constants shall not be used.

Rationale

Octal constants are denoted by a leading zero. Developers can mistake an octal constant as a decimal constant with a redundant leading zero.

Polyspace Implementation

If you use octal constants in a macro definition, the rule checker flags the issue even if the macro is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of octal constants

```
#define CST     021
#define VALUE   010             /* Compliant - constant not used */
#if 010 == 01                   /* Non-Compliant - constant used */
#define CST 021                 /* Non-Compliant - constant not used */
#endif

extern short code[5];
static char* str2 = "abcd\0efg";  /* Compliant */

void main(void) {
    int value1 = 0;              /* Compliant */
    int value2 = 01;             /* Non-Compliant - decimal 01 */
    int value3 = 1;              /* Compliant */
    int value4 = '\109';         /* Compliant */

    code[1] = 109;               /* Compliant     - decimal 109 */
    code[2] = 100;               /* Compliant     - decimal 100 */
    code[3] = 052;               /* Non-Compliant - decimal 42 */
    code[4] = 071;               /* Non-Compliant - decimal 57 */

    if (value1 != CST) {         /* Non-Compliant - decimal 17 */
        value1 = !(value1 != 0);  /* Compliant */
    }
}
```

In this example, the rule is not violated when octal constants are used to define macros `CST` and `VALUE`. The rule is violated only when the macros are used.

Check Information

Group: Literals and Constants
Category: Required
AGC Category: Advisory

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 7.2

A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type

Description

Rule Definition

A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type.

Rationale

The signedness of a constant is determined from:

- Value of the constant.
- Base of the constant: octal, decimal or hexadecimal.
- Size of the various types.
- Any suffixes used.

Unless you use a suffix `u` or `U`, another developer looking at your code cannot determine easily whether a constant is signed or unsigned.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Literals and Constants
Category: Required
AGC Category: Readability

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 7.3

The lowercase character “l” shall not be used in a literal suffix

Description

Rule Definition

The lowercase character “l” shall not be used in a literal suffix.

Rationale

The lowercase character “l” can be confused with the digit “1”. Use the uppercase “L” instead.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Literals and Constants
Category: Required
AGC Category: Readability

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 7.4

A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char"

Description

Rule Definition

A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

Rationale

This rule prevents assignments that allow modification of a string literal.

An attempt to modify a string literal can result in undefined behavior. For example, some implementations can store string literals in read-only memory. An attempt to modify the string literal can result in an exception or crash.

Polyspace Implementation

The rule checker flags assignment of string literals to:

- Pointers with data type other than `const char*`.
- Arrays with data type other than `const char`.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Incorrect Assignment of String Literal

```
char *str1 = "xxxxxx";            /* Non-Compliant */
const char *str2 = "xxxxxx";      /* Compliant */

void checkSystem1(char*);
void checkSystem2(const char*);

void main() {
 checkSystem1("xxxxxx");    /* Non-Compliant */
 checkSystem2("xxxxxx");    /* Compliant */
}
```

In this example, the rule is not violated when string literals are assigned to `const char*` pointers, either directly or through copy of function arguments. The rule is violated only when the `const` qualifier is not used.

Check Information

Group: Literals and Constants
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 11.4` | `MISRA C:2012 Rule 11.8` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.1

Types shall be explicitly specified

Description

Rule Definition

Types shall be explicitly specified.

Rationale

In some circumstances, you can omit types from the C90 standard. In those cases, the `int` type is implicitly specified. However, the omission of an explicit type can lead to confusion. For example, in the declaration `extern void foo (char c, const k);`, the type of `k` is `const int`, but you might expect `const char`.

You might be using an implicit type in:

- Object declarations
- Parameter declarations
- Member declarations
- `typedef` declarations
- Function return types

Polyspace Implementation

The rule checker flags situations where a function parameter or return type is not explicitly specified.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Implicit Types

```
static foo(int a);  /* Non compliant */
static void bar(void);      /* Compliant */
```

In this example, the rule is violated because the return type of `foo` is implicit.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 8.2` | `Check MISRA C:2012 (-misra3)`

Topics
“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.10

An inline function shall be declared with the static storage class

Description

Rule Definition

An inline function shall be declared with the static storage class.

Rationale

If you call an inline function that is declared with external linkage but not defined in the same translation unit, the function might not be inlined. You might not see the reduction in execution time that you expect from inlining.

If you want to make an inline function available to several translation units, you can still define it with the `static` specifier. In this case, place the definition in a header file. Include the header file in all the files where you want the function inlined.

Polyspace Implementation

The rule checker flags definitions that contain the `inline` specifier without an accompanying `static` specifier.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Inlining Functions with External Linkage

```
inline double mult(int val);
inline double mult(int val) {   /* Non compliant */
    return val * 2.0;
}

static inline double div(int val);
static inline double div(int val) {  /* Compliant */
    return val / 2.0;
}
```

In this example, the definition of `mult` is noncompliant because it is inlined without the `static` storage specifier.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 5.9` | `Check MISRA C:2012 (-misra3)`

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.11

When an array with external linkage is declared, its size should be explicitly specified

Description

Rule Definition

When an array with external linkage is declared, its size should be explicitly specified.

Rationale

Although it is possible to declare an array with an incomplete type and access its elements, it is safer to state the size of the array explicitly. If you provide size information for each declaration, a code reviewer can check multiple declarations for their consistency. With size information, a static analysis tool can perform array bounds analysis without analyzing more than one unit.

Polyspace Implementation

The rule checker flags arrays declared with the `extern` specifier if the declaration does not explicitly specify the array size.

Additional Message in Report

Size of array *`array_name`* should be explicitly stated. When an array with external linkage is declared, its size should be explicitly specified.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Array Declarations

```
extern int32_t array1[10];    /*  Compliant  */
extern int32_t array2[];      /*  Non-compliant  */
```

In this example, two arrays are declared `array1` and `array2`. `array1` has external linkage (the `extern` keyword) and a size of 10. `array2` also has external linkage, but no specified size. `array2` is noncompliant because for arrays with external linkage, you must explicitly specify a size.

Check Information

Group: Declarations and Definitions
Category: Advisory
AGC Category: Advisory

See Also

`Check MISRA C:2012 (-misra3)`

Topics
“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.12

Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique

Description

Rule Definition

Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.

Rationale

An implicitly specified enumeration constant has a value one greater than its predecessor. If the first enumeration constant is implicitly specified, then its value is 0. An explicitly specified enumeration constant has the specified value.

If implicitly and explicitly specified constants are mixed within an enumeration list, it is possible for your program to replicate values. Such replications can be unintentional and can cause unexpected behavior.

Polyspace Implementation

The rule checker flags an enumeration if it has an implicitly specified enumeration constant with the same value as another enumeration constant.

Additional Message in Report

The constant *`constant1`* has same value as the constant *`constant2`*.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Replication of Value in Implicitly Specified Enum Constants

```
enum color1 {red_1, blue_1, green_1};   /* Compliant */
enum color2 {red_2 = 1, blue_2 = 2, green_2 = 3};       /* Compliant */
enum color3 {red_3 = 1, blue_3, green_3};     /* Compliant */
enum color4 {red_4, blue_4, green_4 = 1};     /* Non Compliant */
enum color5 {red_5 = 2, blue_5, green_5 = 2};     /* Compliant */
enum color6 {red_6 = 2, blue_6, green_6 = 2, yellow_6};     /* Non Compliant */
```

Compliant situations:

- `color1`: All constants are implicitly specified.
- `color2`: All constants are explicitly specified.
- `color3`: Though there is a mix of implicit and explicit specification, all constants have unique values.
- `color5`: The implicitly specified constants have unique values.

Noncompliant situations:

- `color4`: The implicitly specified constant `blue_4` has the same value as `green_4`.
- `color6`: The implicitly specified constant `blue_6` has the same value as `yellow_6`.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.13

A pointer should point to a const-qualified type whenever possible

Description

Rule Definition

A pointer should point to a const-qualified type whenever possible.

Rationale

This rule ensures that you do not inadvertently use pointers to modify objects.

Polyspace Implementation

The rule checker flags a pointer to a non-`const` function parameter if the pointer does not modify the addressed object. The assumption is that the pointer is not meant to modify the object and so must point to a `const`-qualified type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Pointer That Should Point to `const`-Qualified Types

```
#include <string.h>

typedef unsigned short uint16_t;

uint16_t ptr_ex(uint16_t *p) {      /* Non-compliant */
    return *p;
}

char last_char(char * const s){     /* Non-compliant */
    return s[strlen(s) - 1u];
}

uint16_t first(uint16_t a[5]){      /* Non-compliant */
    return a[0];
}
```

This example shows three different noncompliant pointer parameters.

- In the `ptr_ex` function, `p` does not modify an object. However, the type to which `p` points is not `const`-qualified, so it is noncompliant.
- In `last_char`, the pointer `s` is `const`-qualified but the type it points to is not. This parameter is noncompliant because `s` does not modify an object.
- The function `first` does not modify the elements of the array `a`. However, the element type is not `const`-qualified, so `a` is also noncompliant.

Correction — Use `const` Keywords

One possible correction is to add `const` qualifiers to the definitions.

```
#include <string.h>

typedef unsigned short uint16_t;

uint16_t ptr_ex(const uint16_t *p){     /* Compliant */
    return *p;
}

char last_char(const char * const s){   /* Compliant */
    return s[strlen( s ) - 1u];
}

uint16_t first(const uint16_t a[5]) {   /* Compliant */
    return a[0];
}
```

Check Information

Group: Declarations and Definitions
Category: Advisory
AGC Category: Advisory

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.14

The restrict type qualifier shall not be used

Description

Rule Definition

The restrict type qualifier shall not be used.

Rationale

When you use a `restrict` qualifier carefully, it improves the efficiency of code generated by a compiler. It can also improve static analysis. However, when using the `restrict` qualifier, it is difficult to make sure that the memory areas operated on by two or more pointers do not overlap.

Polyspace Implementation

The rule checker flags all uses of the `restrict` qualifier.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `restrict` Qualifier

```
void f(int n, int * restrict p, int * restrict q)
{
}
```

In this example, both uses of the `restrict` qualifier are flagged.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Advisory

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.2

Function types shall be in prototype form with named parameters

Description

Rule Definition

Function types shall be in prototype form with named parameters.

Rationale

The rule requires that you specify names and data types for all the parameters in a declaration. The parameter names provide useful information regarding the function interface. A mismatch between a declaration and definition can indicate a programming error. For instance, you mixed up parameters when defining the function. By insisting on parameter names, the rule allows a code reviewer to detect this mismatch.

Polyspace Implementation

The rule checker shows a violation if the parameters in a function declaration or definition are missing names or data types.

Additional Message in Report

- Too many arguments to *`function_name`*.
- Too few arguments to *`function_name`*.
- Function types shall be in prototype form with named parameters.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Function Prototype Without Named Parameters

```
extern int func(int);   /* Non compliant */
extern int func2(int n);   /* Compliant */

extern int func3();   /* Non compliant */
extern int func4(void);   /* Compliant */
```

In this example, the declarations of `func` and `func3` are noncompliant because the parameters are missing or do not have names.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 8.1` | `MISRA C:2012 Rule 8.4` | `MISRA C:2012 Rule 17.3` | `Check MISRA C:2012 (-misra3)`

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.3

All declarations of an object or function shall use the same names and type qualifiers

Description

Rule Definition

All declarations of an object or function shall use the same names and type qualifiers.

Rationale

Consistently using parameter names and types across declarations of the same object or function encourages stronger typing. It is easier to check that the same function interface is used across all declarations.

Polyspace Implementation

The rule checker detects situations where parameter names or data types are different between multiple declarations or the declaration and the definition. The checker considers declarations in all translation units and flags issues that are not likely to be detected by a compiler.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Additional Message in Report

- Definition of function *`function_name`* incompatible with its declaration.
- Global declaration of *`function_name`* function has incompatible type with its definition.
- Global declaration of *`variable_name`* variable has incompatible type with its definition.
- All declarations of an object or function shall use the same names and type qualifiers.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Mismatch in Parameter Names

```
extern int div (int num, int den);

int div(int den, int num) { /* Non compliant */
    return(num/den);
}
```

In this example, the rule is violated because the parameter names in the declaration and definition are switched.

Mismatch in Parameter Data Types

```
typedef unsigned short width;
typedef unsigned short height;
```

```
typedef unsigned int area;

extern area calculate(width w, height h);

area calculate(width w, width h) { /* Non compliant *
    return w*h;
}
```

In this example, the rule is violated because the second argument of the `calculate` function has data type:

- `height` in the declaration.
- `width` in the definition.

The rule is violated even though the underlying type of `height` and `width` are identical.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 8.4` | `Check MISRA C:2012 (-misra3)`

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.4

A compatible declaration shall be visible when an object or function with external linkage is defined

Description

Rule Definition

A compatible declaration shall be visible when an object or function with external linkage is defined.

Rationale

If a declaration is visible when an object or function is defined, it allows the compiler to check that the declaration and the definition are compatible.

This rule with `MISRA C:2012 Rule 8.5` enforces the practice of declaring an object (or function) in a header file and including the header file in source files that define or use the object (or function).

Polyspace Implementation

The rule checker detects situations where:

- An object or function is defined without a previous declaration.
- There is a data type mismatch between the object or function declaration and definition. Such a mismatch also causes a compilation error.

The checker now flags tentative definitions (variables declared without an `extern` specifier and not explicitly defined). To avoid the rule violation, declare the variable `static` (defined in one file only), or declare the variable `extern` and follow the declaration with a definition.

Additional Message in Report

- Global definition of *`variable_name`* variable has no previous declaration.
- Function *`function_name`* has no visible compatible prototype at definition.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Definition Without Previous Declaration

Header file:

```
/* file.h */
extern int var2;
void func2(void);
```

Source file:

```
/* file.c */
#include "file.h"
```

```
int var1 = 0;    /* Non compliant */
int var2 = 0;    /* Compliant */

void func1(void) {   /* Non compliant */
}

void func2(void) {   /* Compliant */
}
```

In this example, the definitions of `var1` and `func1` are noncompliant because they are not preceded by declarations.

Mismatch in Parameter Data Types

```
void func(int param1, int param2);

void func(int param1, unsigned int param2) { /* Non compliant */
}
```

In this example, the definition of `func` has a different parameter type from its declaration. The mismatch also causes a compilation error.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 8.2|MISRA C:2012 Rule 8.3|MISRA C:2012 Rule 8.5|MISRA C:2012 Rule 17.3|Check MISRA C:2012 (-misra3)`

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.5

An external object or function shall be declared once in one and only one file

Description

Rule Definition

An external object or function shall be declared once in one and only one file.

Rationale

If you declare an identifier in a header file, you can include the header file in any translation unit where the identifier is defined or used. In this way, you ensure consistency between:

- The declaration and the definition.
- The declarations in different translation units.

The rule enforces the practice of declaring external objects or functions in header files.

Polyspace Implementation

The rule checker checks only explicit `extern` declarations (tentative definitions are ignored). The checker flags variables or functions declared `extern` in a non-header file.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Additional Message in Report

- Object *`object_name`* has external declarations in multiple files.
- Function *`function_name`* has external declarations in multiple files.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Extern Declaration in Non-Header File

Header file:

```
/* file.h */
extern int var;
extern void func1(void);   /* Compliant */
```

Source file:

```
/* file.c */
#include "file.h"

extern void func2(void);   /* Non compliant */
```

```
/* Definitions */
int var = 0;
void func1(void) {}
```

In this example, the declaration of external function `func2` is noncompliant because it occurs in a non-header file. The other external object and function declarations occur in a header file and comply with this rule.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 8.4` | `Check MISRA C:2012 (-misra3)`

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.6

An identifier with external linkage shall have exactly one external definition

Description

Rule Definition

An identifier with external linkage shall have exactly one external definition.

Rationale

If you use an identifier for which multiple definitions exist in different files or no definition exists, the behavior is undefined.

Multiple definitions in different files are not permitted by this rule even if the definitions are the same.

Polyspace Implementation

The checker flags multiple definitions only if the definitions occur in different files.

The checker does not consider tentative definitions as definitions. For instance, the following code does not violate the rule:

```
int val;
int val=1;
```

The checker does not show a violation if a function is not defined at all but declared with external linkage and called in the source code.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Additional Message in Report

- Forbidden multiple definitions for function *`function_name`*.
- Forbidden multiple tentative definitions for object *`object_name`*.
- Global variable *`variable_name`* multiply defined.
- Function *`function_name`* multiply defined.
- Global variable has multiple tentative definitions.
- Undefined global variable *`variable_name`*.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Variable Multiply Defined

First source file:

```
extern int var = 1;
```

Second source file:

```
int var = 0;   /* Non compliant */
```

In this example, the global variable `var` is multiply defined. Unless explicitly specified with the `static` qualifier, the variables have external linkage.

Function Multiply Defined

Header file:

```
/* file.h */
int func(int param);
```

First source file:

```
/* file1.c */
#include "file.h"

int func(int param) {
    return param+1;
}
```

Second source file:

```
/* file2.c */
#include "file.h"

int func(int param) {  /* Non compliant */
    return param-1;
}
```

In this example, the function `func` is multiply defined. Unless explicitly specified with the `static` qualifier, the functions have external linkage.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”

“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.7

Functions and objects should not be defined with external linkage if they are referenced in only one translation unit

Description

Rule Definition

Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.

Rationale

Compliance with this rule avoids confusion between your identifier and an identical identifier in another translation unit or library. If you restrict or reduce the visibility of an object by giving it internal linkage or no linkage, you or someone else is less likely to access the object inadvertently.

Polyspace Implementation

The rule checker flags:

- Objects that are defined at file scope without the `static` specifier but used only in one file.
- Functions that are defined without the `static` specifier but called only in one file.

If you intend to use the object or function in one file only, declare it static.

If your code does not contain a `main` function and you use options such as `Variables to initialize (-main-generator-writes-variables)` with value `custom` to explicitly specify a set of variables to initialize, the checker does not flag those variables. The checker assumes that in a real application, the file containing the `main` must initialize the variables in addition to any file that currently uses them. Therefore, the variables are used in more than one translation unit.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Additional Message in Report

- Variable *`variable_name`* should have internal linkage.
- Function *`function_name`* should have internal linkage.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Variable with External Linkage Used in One File

Header file:

```
/* file.h */
extern int var;
```

First source file:

```
/* file1.c */
#include "file.h"

int var;    /* Compliant */
int var2;   /* Non compliant */
static int var3; /* Compliant */

void reset(void);

void reset(void) {
    var = 0;
    var2 = 0;
    var3 = 0;
}
```

Second source file:

```
/* file2.c */
#include "file.h"

void increment(int var2);

void increment(int var2) {
    var++;
    var2++;
}
```

In this example:

- The declaration of `var` is compliant because `var` is declared with external linkage and used in multiple files.
- The declaration of `var2` is noncompliant because `var2` is declared with external linkage but used in one file only.

  It might appear that `var2` is defined in both files. However, in the second file, `var2` is a parameter with no linkage and is not the same as the `var2` in the first file.
- The declaration of `var3` is compliant because `var3` is declared with internal linkage (with the `static` specifier) and used in one file only.

Function with External Linkage Used in One File

Header file:

```
/* file.h */
extern int var;
extern void increment1 (void);
```

First source file:

```
/* file1.c */
#include "file.h"
```

```
int var;

void increment2(void);
static void increment3(void);
void func(void);

void increment2(void) { /* Non compliant */
    var+=2;
}

static void increment3(void) { /* Compliant */
    var+=3;
}

void func(void) {
    increment1();
    increment2();
    increment3();
}
```

Second source file:

```
/* file2.c */
#include "file.h"

void increment1(void) { /* Compliant */
    var++;
}
```

In this example:

- The definition of `increment1` is compliant because `increment1` is defined with external linkage and called in a different file.
- The declaration of `increment2` is noncompliant because `increment2` is defined with external linkage but called in the same file and nowhere else.
- The declaration of `increment3` is compliant because `increment3` is defined with internal linkage (with the `static` specifier) and called in the same file and nowhere else.

Check Information

Group: Declarations and Definitions
Category: Advisory
AGC Category: Advisory

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.8

The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage

Description

Rule Definition

The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.

Rationale

If you do not use the `static` specifier consistently in all declarations of objects with internal linkage, you might declare the same object with external and internal linkage.

In this situation, the linkage follows the earlier specification that is visible (C99 Standard, Section 6.2.2). For instance, if the earlier specification indicates internal linkage, the object has internal linkage even though the latter specification indicates external linkage. If you notice the latter specification alone, you might expect otherwise.

Polyspace Implementation

The rule checker detects situations where:

- The same object is declared multiple times with different storage specifiers.
- The same function is declared and defined with different storage specifiers.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Linkage Conflict Between Variable Declarations

```
static int foo = 0;
extern int foo;         /* Non-compliant */

extern int hhh;
static int hhh;         /* Non-compliant */
```

In this example, the first line defines `foo` with internal linkage. The first line is compliant because the example uses the `static` keyword. The second line does not use `static` in the declaration, so the declaration is noncompliant. By comparison, the third line declares `hhh` with an `extern` keyword creating external linkage. The fourth line declares `hhh` with internal linkage, but this declaration conflicts with the first declaration of `hhh`.

Correction — Consistent `static` and `extern` Use

One possible correction is to use `static` and `extern` consistently:

```
static int foo = 0;
static int foo;

extern int hhh;
extern int hhh;
```

Linkage Conflict Between Function Declaration and Definition

```
static int fee(void);  /* Compliant - declaration: internal linkage */
int fee(void){         /* Non-compliant */
  return 1;
}

static int ggg(void);  /* Compliant - declaration: internal linkage */
extern int ggg(void){  /* Non-compliant */
  return 1 + x;
}
```

This example shows two internal linkage violations. Because `fee` and `ggg` have internal linkage, you must use a `static` class specifier to be compliant with MISRA.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 8.9

An object should be defined at block scope if its identifier only appears in a single function

Description

Rule Definition

An object should be defined at block scope if its identifier only appears in a single function.

Rationale

If you define an object at block scope, you or someone else is less likely to access the object inadvertently outside the block.

Polyspace Implementation

The rule checker flags `static` objects that are accessed in one function only but declared at file scope.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Object Declared at File Scope but Used in One Function

```
static int ctr;   /* Non compliant */

int checkStatus(void);
void incrementCount(void);

void incrementCount(void) {
    ctr=0;
    while(1) {
        if(checkStatus())
            ctr++;
    }
}
```

In this example, the declaration of `ctr` is noncompliant because it is declared at file scope but used only in the function `incrementCount`. Declare `ctr` in the body of `incrementCount` to be MISRA C-compliant.

Check Information

Group: Declarations and Definitions
Category: Advisory
AGC Category: Advisory

See Also

`Check MISRA C:2012 (-misra3)`

Topics
“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 9.1

The value of an object with automatic storage duration shall not be read before it has been set

Description

Message in Report:

Rule Definition

The value of an object with automatic storage duration shall not be read before it has been set.

Rationale

A variable with an automatic storage duration is allocated memory at the beginning of an enclosing code block and deallocated at the end. All non-global variables have this storage duration, except those declared `static` or `extern`.

Variables with automatic storage duration are not automatically initialized and have indeterminate values. Therefore, you must not read such a variable before you have set its value through a write operation.

Polyspace Implementation

The Polyspace analysis checks some of the violations as non-initialized variables. For more information, see `Non-initialized variable`.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results. In Code Prover, you can also see a difference in results based on your choice for the option . See “Check for Coding Standard Violations”.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Initialization
Category: Mandatory
AGC Category: Mandatory

See Also

`MISRA C:2012 Rule 15.1|MISRA C:2012 Rule 15.3|Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 9.2

The initializer for an aggregate or union shall be enclosed in braces

Description

Rule Definition

The initializer for an aggregate or union shall be enclosed in braces.

Rationale

The rule applies to both objects and subobjects. For example, when initializing a structure that contains an array, the values assigned to the structure must be enclosed in braces. Within these braces, the values assigned to the array must be enclosed in another pair of braces.

Enclosing initializers in braces improves clarity of code that contains complex data structures such as multidimensional arrays and arrays of structures.

Tip To avoid nested braces for subobjects, use the syntax `{0}`, which sets all values to zero.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Initialization of Two-dimensional Arrays

```
void initialize(void) {
    int x[4][2] = {{0,0},{1,0},{0,1},{1,1}}; /* Compliant */
    int y[4][2] = {{0},{1,0},{0,1},{1,1}};   /* Compliant */
    int z[4][2] = {0};                       /* Compliant */
    int w[4][2] = {0,0,1,0,0,1,1,1};         /* Non-compliant */
}
```

In this example, the rule is not violated when:

- Initializers for each row of the array are enclosed in braces.
- The syntax `{0}` initializes all elements to zero.

The rule is violated when a separate pair of braces is not used to enclose the initializers for each row.

Check Information

Group: Initialization
Category: Required
AGC Category: Readability

See Also

`Check MISRA C:2012 (-misra3)`

Topics
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 9.3

Arrays shall not be partially initialized

Description

Rule Definition

Arrays shall not be partially initialized.

Rationale

Providing an explicit initialization for each array element makes it clear that every element has been considered.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Partial and Complete Initializations

```
void func(void) {
    int x[3] = {0,1,2};              /* Compliant */
    int y[3] = {0,1};                /* Non-compliant */
    int z[3] = {0};                  /* Compliant - exception */
    int a[30] = {[1] = 1,[15]=1};    /* Compliant - exception */
    int b[30] = {{1} = 1, 1};        /* Non-compliant */
    char c[20] = "Hello World";      /* Compliant - exception */
}
```

In this example, the rule is not violated when each array element is explicitly initialized.

The rule is violated when some elements of the array are implicitly initialized. Exceptions include the following:

- The initializer has the form `{0}`, which initializes all elements to zero.
- The array initializer consists *only* of designated initializers. Typically, you use this approach for sparse initialization.
- The array is initialized using a string literal.

Check Information

Group: Initialization
Category: Required
AGC Category: Readability

See Also

`Check MISRA C:2012 (-misra3)`

Topics
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 9.4

An element of an object shall not be initialized more than once

Description

Rule Definition

An element of an object shall not be initialized more than once.

Rationale

Designated initializers allow explicitly initializing elements of objects such as arrays in any order. However, using designated initializers, one can inadvertently initialize the same element twice and therefore overwrite the first initialization.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Array Initialization Using Designated Initializers

```
void func(void) {
    int a[5] = {-2,-1,0,1,2};               /* Compliant */
    int b[5] = {[0]=-2, [1]=-1, [2]=0, [3]=1, [4]=2};
                                            /* Compliant */
    int c[5] = {[0]=-2, [1]=-1, [1]=0, [3]=1, [4]=2};
                                            /* Non-compliant */
}
```

In this example, the rule is violated when the array element `c[1]` is initialized twice using a designated initializer.

Structure Initialization Using Designated Initializers

```
struct myStruct {
    int a;
    int b;
    int c;
    int d;
};

void func(void) {
    struct myStruct struct1 = {-4,-2,2,4};   /* Compliant */
    struct myStruct struct2 = {.a=-4, .b=-2, .c=2, .d=4};
                                             /* Compliant */
    struct myStruct struct3 = {.a=-4, .b=-2, .b=2, .d=4};
                                             /* Non-compliant */
}
```

In this example, the rule is violated when `struct3.b` is initialized twice using a designated initializer.

Check Information

Group: Initialization
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 9.5

Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly

Description

Rule Definition

Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

Rationale

If the size of an array is not specified explicitly, it is determined by the highest index of the elements that are initialized. When using long designated initializers, it might not be immediately apparent which element has the highest index.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Using Designated Initializers Without Specifying Array Size

```
int a[5] = {[0]= 1, [2] = 1, [4]= 1, [1] = 1};        /* Compliant */
int b[] = {[0]= 1, [2] = 1, [4]= 1, [1] = 1};         /* Non-compliant */
int c[] = {[0]= 1, [1] = 1, [2]= 1, [3]=0, [4] = 1};  /* Non-compliant */

void display(int);

void main() {
    func(a,5);
    func(b,5);
    func(c,5);
}

void func(int* arr, int size) {
    for(int i=0; i<size; i++)
        display(arr[i]);
}
```

In this example, the rule is violated when the arrays `b` and `c` are initialized using designated initializers but the array size is not specified.

Check Information

Group: Initialization
Category: Required
AGC Category: Readability

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Dir 1.1

Any implementation-defined behavior on which the output of the program depends shall be documented and understood

Description

Directive Definition

Any implementation-defined behavior on which the output of the program depends shall be documented and understood.

Rationale

A code construct has implementation-defined behavior if the C standard allows compilers to choose their own specifications for the construct. The full list of implementation-defined behavior is available in Annex J.3 of the standard ISO/IEC 9899:1999 (C99) and in Annex G.3 of the standard ISO/IEC 9899:1990 (C90).

If you understand and document all implementation-defined behavior, you can be assured that all output of your program is intentional and not produced by chance.

Polyspace Implementation

The analysis detects the following possibilities of implementation-defined behavior in C99 and their counterparts in C90. If you know the behavior of your compiler implementation, justify the analysis result with appropriate comments. To justify a result, assign one of these statuses: `Justified`, `No action planned`, or `Not a defect`.

Tip To mass-justify all results that indicate the same implementation-defined behavior, use the **Detail** column on the **Results List** pane. Click the column header so that all results with the same entry are grouped together. Select the first result and then select the last result while holding the `Shift` key. Assign a status to one of the results. If you do not see the **Detail** column, right-click any other column header and enable this column.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps

J.3.2: Environment	An alternative manner in which `main` function may be defined.	The analysis flags `main` with arguments and return types other than:
		`int main(void) { ... }`
		or
		`int main(int argc, char *argv[]) { ... }`
		See section 5.1.2.2.1 of the C99 Standard.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps
J.3.2: Environment	The set of environment names and the method for altering the environment list used by the `getenv` function.	The analysis flags uses of the `getenv` function. For this function, you need to know the list of environment variables and how the list is modified. See section 7.20.4.5 of the C99 Standard.
J.3.6: Floating Point	The rounding behaviors characterized by non-standard values of `FLT_ROUNDS`.	The analysis flags the include of `float.h` if values of `FLT_ROUNDS` are outside the set, {-1, 0, 1, 2, 3}. Only the values in this set lead to well-defined rounding behavior. See section 5.2.4.2.2 of the C99 Standard.
J.3.6: Floating Point	The evaluation methods characterized by non-standard negative values of `FLT_EVAL_METHOD`.	The analysis flags the include of `float.h` if values of `FLT_EVAL_METHOD` are outside the set, {-1, 0, 1, 2}. Only the values in this set lead to well-defined behavior for floating-point operations. See section 5.2.4.2.2 of the C99 Standard.
J.3.6: Floating Point	The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value.	The analysis flags conversions from integer to floating-point data types of smaller size (for example, 64-bit `int` to 32-bit `float`). See section 6.3.1.4 of the C99 Standard.
J.3.6: Floating Point	The direction of rounding when a floating-point number is converted to a narrower floating-point number.	The analysis flags these conversions: • `double` to `float` • `long double` to `double` or `float` See section 6.3.1.5 of the C99 Standard.

J.3.6: Floating Point	The default state for the `FENV_ACCESS` pragma.	The analysis flags use of the pragma other than:
		`#pragma STDC FENV_ACCESS ON`
		or
		`#pragma STDC FENV_ACCESS OFF`
		
		See section 7.6.1 of the C99 Standard.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps

J.3.6: Floating Point	The default state for the `FP_CONTRACT` pragma.	The analysis flags use of the pragma other than:
		`#pragma STDC FP_CONTRACT ON`
		or
		`#pragma STDC FP_CONTRACT OFF`
		See section 7.12.2 of the C99 Standard.

J.3.11: Preprocessing Directives	The behavior on each recognized non-STDC #pragma directive.	The analysis flags the pragma usage: `#pragma` *`pp-tokens`* where the processing token `STDC` does not immediately follow`pragma`. For instance: `#pragma FENV_ACCESS ON` See section 6.10.6 of the C99 Standard.
J.3.12: Library Functions	Whether the `feraiseexcept` function raises the ‘‘inexact’’ floating-point exception in addition to the ‘‘overflow’’ or ‘‘underflow’’ floating-point exception.	The analysis flags calls to the `feraiseexcept` function. See section 7.6.2.3 of the C99 Standard.
J.3.12: Library Functions	Strings other than `"C"` and `""` that may be passed as the second argument to the `setlocale` function.	The analysis flags calls to the `setlocale` function when its second argument is not `"C"` or `""`. See section 7.11.1.1 of the C99 Standard.
J.3.12: Library Functions	The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 or greater than 2.	The analysis flags the include of `math.h` if `FLT_EVAL_METHOD` has values outside the set {0,1,2}. See section 7.12 of the C99 Standard.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps
J.3.12: Library Functions	The base-2 logarithm of the modulus used by the `remquo` functions in reducing the quotient.	The analysis flags calls to the `remquo`, `remquof` and `remquol` function. See section 7.12.10.3 of the C99 Standard.
J.3.12: Library Functions	The termination status returned to the host environment by the `abort`, `exit`, or `_Exit` function.	The analysis flags calls to the `abort`, `exit`, or `_Exit` function. See sections 7.20.4.1, 7.20.4.3 or 7.20.4.4 of the C99 Standard.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: The implementation
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017b

MISRA C:2012 Dir 2.1

All source files shall compile without any compilation errors

Description

Directive Definition

All source files shall compile without any compilation errors.

Rationale

A conforming compiler is permitted to produce an object module despite the presence of compilation errors. However, execution of the resulting program can produce unexpected behavior.

Polyspace Implementation

The software raises a violation of this directive if it finds a compilation error. Because Code Prover is more strict about compilation errors compared to Bug Finder, the coding rules checking in the two products can produce different results for this directive.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Compilation and build
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)|MISRA C:2012 Rule 1.1`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Dir 4.1

Run-time failures shall be minimized

Description

Directive Definition

Run-time failures shall be minimized.

Rationale

Some areas to concentrate on are:

- Arithmetic errors
- Pointer arithmetic
- Array bound errors
- Function parameters
- Pointer dereferencing
- Dynamic memory

Polyspace Implementation

This directive is checked through the Polyspace analysis. For more information, see:

- “Defects”.
- “Run-Time Checks” (Polyspace Code Prover).

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Code design
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Dir 4.11|MISRA C:2012 Rule 1.3|MISRA C:2012 Rule 18.1|MISRA C:2012 Rule 18.2|MISRA C:2012 Rule 18.3|Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Dir 4.10

Precautions shall be taken in order to prevent the contents of a header file being included more than once

Description

Directive Definition

Precautions shall be taken in order to prevent the contents of a header file being included more than once.

Rationale

When a translation unit contains a complex hierarchy of nested header files, it is possible for a particular header file to be included more than once, leading to confusion. If this multiple inclusion produces multiple or conflicting definitions, then your program can have undefined or erroneous behavior.

For instance, suppose that a header file contains:

```
#ifdef _WIN64
   int env_var;
#elseif
   long int env_var;
#endif
```

If the header file is contained in two inclusion paths, one that defines the macro `_WIN64` and another that undefines it, you can have conflicting definitions of `env_var`.

Polyspace Implementation

If you include a header file whose contents are not guarded from multiple inclusion, the analysis raises a violation of this directive. The violation is shown at the beginning of the header file.

You can guard the contents of a header file from multiple inclusion by using one of the following methods:

```
<start-of-file>
#ifndef <control macro>
#define <control macro>
    /* Contents of file */
#endif
<end-of-file>
```

or

```
<start-of-file>
#ifdef <control macro>
#error ...
#else
#define <control macro>
```

```
    /* Contents of file */
#endif
<end-of-file>
```

Unless you use one of these methods, Polyspace flags the header file inclusion as noncompliant.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Code After Macro Guard

```
#ifndef  __MY_MACRO__
#define __MY_MACRO__
    void func(void);
#endif
void func2(void);
```

If a header file contains this code, it is noncompliant because the macro guard does not cover the entire content of the header file. The line `void func2(void)` is outside the guard.

Note You can have comments outside the macro guard.

Code Before Macro Guard

```
void func(void);
#ifndef  __MY_MACRO__
#define __MY_MACRO__
    void func2(void);
#endif
```

If a header file contains this code, it is noncompliant because the macro guard does not cover the entire content of the header file. The line `void func(void)` is outside the guard.

Note You can have comments outside the macro guard.

Mismatch in Macro Guard

```
#ifndef  __MY_MACRO__
#define __MY_MARCO__
    void func(void);
    void func2(void);
#endif
```

If a header file contains this code, it is noncompliant because the macro name in the `#ifndef` statement is different from the name in the following `#define` statement.

Check Information

Group: Code Design

Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Dir 4.11

The validity of values passed to library functions shall be checked

Description

Directive Definition

The validity of values passed to library functions shall be checked.

Rationale

Many Standard C functions do not check the validity of parameters passed to them. Even if checks are performed by a compiler, there is no guarantee that the checks are adequate. For example, you should not pass negative numbers to `sqrt` or `log`.

Polyspace Implementation

Polyspace raises a violation result for library function arguments if the following are all true:

- Argument is a local variable.
- Local variable is not tested between last assignment and call to the library function.
- Corresponding parameter of the library function has a restricted input domain.
- Library function is one of the following common mathematical functions:
  - `sqrt`
  - `tan`
  - `pow`
  - `log`
  - `log10`
  - `fmod`
  - `acos`
  - `asin`
  - `acosh`
  - `atanh`
  - or `atan2`

Bug Finder and Code Prover check this rule differently. The analysis can produce different results.

Tip To mass-justify all results related to the same library function, use the **Detail** column on the **Results List** pane. Click the column header so that all results with the same entry are grouped together. Select the first result and then select the last result while holding the `Shift` key. Assign a status to one of the results. If you do not see the **Detail** column, right-click any other column header and enable this column.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Code design
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Dir 4.1` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Dir 4.13

Functions which are designed to provide operations on a resource should be called in an appropriate sequence

Description

Directive Definition

Functions which are designed to provide operations on a resource should be called in an appropriate sequence.

Rationale

You typically use functions operating on a resource in the following way:

1. You allocate the resource.

   For example, you open a file or critical section.
2. You use the resource.

   For example, you read from the file or perform operations in the critical section.
3. You deallocate the resource.

   For example, you close the file or critical section.

For your functions to operate as you expect, perform the steps in sequence. For instance, if you call a resource allocation function on a certain execution path, you must call a deallocation function on that path.

Polyspace Implementation

Polyspace Bug Finder detects a violation of this rule if you specify multitasking options and your code contains one of these defects:

- `Missing lock`: A task calls an unlock function before calling the corresponding lock function.
- `Missing unlock`: A task calls a lock function but ends without a call to the corresponding unlock function.
- `Double lock`: A task calls a lock function twice without an intermediate call to an unlock function.
- `Double unlock`: A task calls an unlock function twice without an intermediate call to a lock function.

For more information on the multitasking options, see “Multitasking”.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Multitasking: Lock Function That Is Missing Unlock Function

```
typedef signed int int32_t;
typedef signed short int16_t;

typedef struct tag_mutex_t {
    int32_t value;
} mutex_t;


extern mutex_t mutex_lock   ( void );
extern void  mutex_unlock (  mutex_t m );

extern int16_t x;
void func(void);

void task1(void) {
     func();
}

void task2(void) {
     func();
}

void func ( void ) {
    mutex_t m = mutex_lock ( );  /* Non-compliant */

    if ( x > 0 )  {
        mutex_unlock ( m );
    }  else  {
        /* Mutex not unlocked on this path */
    }
}
```

In this example, the rule is violated when:

- You specify that the functions `mutex_lock` and `mutex_unlock` are paired.

  `mutex_lock` begins a critical section and `mutex_unlock` ends it.
- The function `mutex_lock` is called. However, if `x <= 0`, the function `mutex_unlock` is not called.

To enable detection of this rule violation, you must specify these analysis options.

Option	Specification	
Configure multitasking manually	☑	
Entry points	`task1` `task2`	
Critical section details	**Starting routine**	**Ending routine**
	`mutex_lock`	`mutex_unlock`

For more information on the options, see:

- `Tasks (-entry-points)`
- `Critical section details (-critical-section-begin -critical-section-end)`

Check Information

Group: Code design
Category: Advisory
AGC Category: Advisory

See Also

`Check MISRA C:2012 (-misra3)` | `MISRA C:2012 Rule 22.1` | `MISRA C:2012 Rule 22.2` | `MISRA C:2012 Rule 22.6`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Dir 4.14

The validity of values received from external sources shall be checked

Description

Directive Definition

The validity of values received from external sources shall be checked.

Rationale

The values originating from external sources can be invalid because of errors or deliberate modification by attackers. Before using the data, you must check the data for validity.

For instance:

- Before using an external input as array index, you must check if it can potentially cause an array bounds error.
- Before using a variable to control a loop, you must check if it can potentially result in an infinite loop.

Polyspace Implementation

The checker for this rule looks for the same issues as these defect checkers:

- `Array access with tainted index`
- `Command executed from externally controlled path`
- `Execution of externally controlled command`
- `Host change using externally controlled elements`
- `Library loaded from externally controlled path`
- `Loop bounded with tainted value`
- `Memory allocation with tainted size`
- `Pointer dereference with tainted offset`
- `Tainted division operand`
- `Tainted modulo operand`
- `Tainted NULL or non-null-terminated string`
- `Tainted sign change conversion`
- `Tainted size of variable length array`
- `Tainted string format`
- `Use of externally controlled environment variable`
- `Use of tainted pointer`

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Validity of External Values Not Checked

```
#include <stdio.h>

void f1(char from_user[])
{
        char input [128];
        (void) sscanf (from_user, "%128c", input);
        (void) sprintf ("%s", input);
}
```

In this example, the `sscanf` statement is noncompliant as there is no check to ensure that the user input is null terminated. The subsequent `sprintf` statement that outputs the string can potentially lead to an array bounds error (buffer overrun).

Check Information

Group: Code design
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Dir 4.3

Assembly language shall be encapsulated and isolated

Description

Directive Definition

Assembly language shall be encapsulated and isolated.

Rationale

Encapsulating assembly language is beneficial because:

- It improves readability.
- The name, and documentation, of the encapsulating macro or function makes the intent of the assembly language clear.
- All uses of assembly language for a given purpose can share encapsulation, which improves maintainability.
- You can easily substitute the assembly language for a different target or for purposes of static analysis.

Polyspace Implementation

Polyspace does not raise a warning on assembly language code encapsulated in the following:

- `asm` functions or `asm` pragmas
- Macros

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Assembly Language Code in C Function

```
enum boolVal {TRUE, FALSE};
enum boolVal isTaskActive;
void taskHandler(void);

void taskHandler(void) {
    isTaskActive = FALSE;
    // Software interrupt for task switching
    asm volatile
    (
        "SWI &02"     /* Service #1: calculate run-time */
    );
    return;
}
```

In this example, the rule violation occurs because the assembly language code is embedded directly in a C function `taskHandler` that contains other C language statements.

Correction: Encapsulate Assembly Code in Macro

One possible correction is to encapsulate the assembly language code in a macro and invoke the macro in the function `taskHandler`.

```
#define  RUN_TIME_CALC \
asm volatile \
    ( \
        "SWI &02"     /* Service #1: calculate run-Time */ \
    )\

enum boolVal {TRUE, FALSE};
enum boolVal isTaskActive;
void taskHandler(void);

void taskHandler(void) {
    isTaskActive = FALSE;
    RUN_TIME_CALC;
    return;
}
```

Check Information

Group: Code design
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 1.2` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Dir 4.5

Identifiers in the same name space with overlapping visibility should be typographically unambiguous

Description

Directive Definition

Identifiers in the same name space with overlapping visibility should be typographically unambiguous.

Rationale

What “unambiguous” means depends on the alphabet and language in which source code is written. When you use identifiers that are typographically close, you can confuse between them.

For the Latin alphabet as used in English words, at a minimum, the identifiers should not differ by:

- The interchange of a lowercase letter with its uppercase equivalent.
- The presence or absence of the underscore character.
- The interchange of the letter `O` and the digit `0`.
- The interchange of the letter `I` and the digit `1`.
- The interchange of the letter `I` and the letter `l`.
- The interchange of the letter `S` and the digit `5`.
- The interchange of the letter `Z` and the digit `2`.
- The interchange of the letter `n` and the letter `h`.
- The interchange of the letter `B` and the digit `8`.
- The interchange of the letters `rn` and the letter `m`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Typographically Ambiguous Identifiers

```
void func(void) {
    int id1_numval;
    int id1_num_val;  /* Non-compliant */

    int id2_numval;
    int id2_numVal;   /* Non-compliant */

    int id3_lvalue;
    int id3_Ivalue;   /* Non-compliant */

    int id4_xyz;
    int id4_xy2;      /* Non-compliant */
```

```
    int id5_zer0;
    int id5_zer0;     /* Non-compliant */

    int id6_rn;
    int id6_m;        /* Non-compliant */
}
```

In this example, the rule is violated when identifiers that can be confused for each other are used.

Check Information

Group: Code design
Category: Advisory
AGC Category: Readability

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

MISRA C:2012 Dir 4.6

`typedefs` that indicate size and signedness should be used in place of the basic numerical types

Description

Directive Definition

`typedefs` that indicate size and signedness should be used in place of the basic numerical types.

Rationale

When the amount of memory being allocated is important, using specific-length types makes it clear how much storage is being reserved for each object.

Polyspace Implementation

The rule checker flags use of basic data types in variable or function declarations and definitions. The rule enforces use of `typedef`s instead.

The rule checker does not flag the use of basic types in the `typedef` statements themselves.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Direct Use of Basic Types in Definitions

```
typedef unsigned int uint32_t;

int x = 0;       /* Non compliant */
uint32_t y = 0;  /* Compliant */
```

In this example, the declaration of `x` is noncompliant because it uses a basic type directly.

Check Information

Group: Code design
Category: Advisory
AGC Category: Advisory

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Dir 4.7

If a function returns error information, then that error information shall be tested

Description

Directive Definition

If a function returns error information, then that error information shall be tested.

Rationale

Typically a function indicates whether an error occurred during execution, via a special return value or by another means.

If a function provides a mechanism to determine errors, before you use the function return value, you must check for such errors.

Polyspace Implementation

The checking of this directive follows the same specifications as the defect checker `Returned value of a sensitive function not checked`.

This directive is only partially supported.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Code design
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

MISRA C:2012 Dir 4.8

If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden

Description

Rule Definition

If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden.

Rationale

If a pointer to a structure or union is not dereferenced in a file, the implementation details of the structure or union need not be available in the translation unit for the file. You can hide the implementation details such as structure members and protect them from unintentional changes.

Define an opaque type that can be referenced via pointers but whose contents cannot be accessed.

Polyspace Implementation

If a structure or union is defined in a file or a header file included in the file, a pointer to this structure or union declared but the pointer never dereferenced in the file, the checker flags a coding rule violation. The structure or union definition should not be visible to this file.

If you see a violation of this rule on a structure definition, identify if you have defined a pointer to the structure in the same file or in a header file included in the file. Then check if you dereference the pointer anywhere in the file. If you do not dereference the pointer, the structure definition should be hidden from this file and included header files.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Object Implementation Revealed

`file.h`: Contains structure implementation.

```
#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct  {
  int a;
} myStruct;

#endif
```

`file.c`: Includes `file.h` but does not dereference structure.

```
#include "file.h"
```

```
myStruct* getObj(void);
void useObj(myStruct*);

void func() {
  myStruct *sPtr = getObj();
  useObj(sPtr);
}
```

In this example, the pointer to the type `myStruct` is not dereferenced. The pointer is simply obtained from the `getObj` function and passed to the `useObj` function.

The implementation of `myStruct` is visible in the translation unit consisting of `file.c` and `file.h`.

Correction — Define Opaque Type

One possible correction is to define an opaque data type in the header file `file.h`. The opaque data type `ptrMyStruct` points to the `myStruct` structure without revealing what the structure contains. The structure `myStruct` itself can be defined in a separate translation unit, in this case, consisting of the file `file2.c`. The common header file `file.h` must be included in both `file.c` and `file2.c` for linking the structure definition to the opaque type definition.

`file.h`: Does not contain structure implementation.

```
#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct myStruct *ptrMyStruct;

ptrMyStruct getObj(void);
void useObj(ptrMyStruct);

#endif
```

`file.c`: Includes `file.h` but does not dereference structure.

```
#include "file.h"

void func() {
  ptrMyStruct sPtr = getObj();
  useObj(sPtr);
}
```

`file2.c`: Includes `file.h` and dereferences structure.

```
#include "file.h"

struct myStruct {
  int a;
};

void useObj(ptrMyStruct ptr) {
    (ptr->a)++;
}
```

Check Information

Group: Code design

Category: Advisory
AGC Category: Advisory

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2018a

MISRA C:2012 Dir 4.9

A function should be used in preference to a function-like macro where they are interchangeable

Description

Directive Definition

A function should be used in preference to a function-like macro where they are interchangeable.

Rationale

In most circumstances, use functions instead of macros. Functions perform argument type-checking and evaluate their arguments once, avoiding problems with potential multiple side effects.

Polyspace Implementation

Polyspace considers all function-like macro definitions.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Code design
Category: Advisory
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 13.2` | `MISRA C:2012 Rule 20.7` | `Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Dir 4.12

Dynamic memory allocation shall not be used

Description

Rule Definition

Dynamic memory allocation shall not be used.

Rationale

Using dynamic memory allocation and deallocation routines provided by the Standard Library or third-party libraries can cause undefined behavior. For instance:

- You use `free` to deallocate memory that you did not allocate with `malloc`, `calloc`, or `realloc`.
- You use a pointer that points to a freed memory location.
- You access allocated memory that has no value stored into it.

Dynamic memory allocation and deallocation routines from third-party libraries are likely to exhibit similar undefined behavior.

If you choose to use dynamic memory allocation and deallocation routines, ensure that your program behavior is predictable. For example, ensure that you safely handle allocation failure due to insufficient memory.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `malloc`, `calloc`, `realloc` and `free`

```
#include <stdlib.h>

static int foo(void);

typedef struct struct_1 {
    int a;
    char c;
} S_1;

static int foo(void) {

    S_1 * ad_1;
    int  * ad_2;
    int  * ad_3;

    ad_1 = (S_1*)calloc(100U, sizeof(S_1));        /* Non-compliant */
    ad_2 = malloc(100U * sizeof(int));             /* Non-compliant */
    ad_3 = realloc(ad_3, 60U * sizeof(long));      /* Non-compliant */
```

```
    free(ad_1);                                 /* Non-compliant */
    free(ad_2);                                 /* Non-compliant */
    free(ad_3);                                 /* Non-compliant */

    return 1;
}
```

In this example, the rule is violated when the functions `malloc`, `calloc`, `realloc` and `free` are used.

Check Information

Group: Code Design
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

“Check for Coding Standard Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2019b

MISRA C++: 2008

MISRA C++:2008 Rule 0-1-1

A project shall not contain unreachable code

Description

Rule Definition

A project shall not contain unreachable code.

Rationale

This rule flags situations where a group of statements is unreachable because of syntactic reasons. For instance, code following a `return` statement are always unreachable.

Unreachable code involve unnecessary maintenance and can often indicate programming errors.

Polyspace Implementation

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Unreachable statements

```
int func(int arg) {
 int temp = 0;
 switch(arg) {
     temp = arg; // Noncompliant
     case 1:
     {
         break;
     }
     default:
     {
         break;
     }
 }
 return arg;
 arg++; // Noncompliant
}
```

These statements are unreachable:

- Statements inside a `switch` statement that do not belong to a `case` or `default` block.
- Statements after a `return` statement.

Check Information

Group: Language Independent Issues
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 0-1-2

A project shall not contain infeasible paths

Description

Rule Definition

A project shall not contain infeasible paths.

Rationale

This rule flags situations where a group of statements is redundant because of nonsyntactic reasons. For instance, an `if` condition is always true or false. Code that is unreachable from syntactic reasons are flagged by rule 0-1-1.

Unreachable or redundant code involve unnecessary maintenance and can often indicate programming errors.

Polyspace Implementation

Bug Finder and Code Prover check this rule differently. The analysis can produce different results.

- Bug Finder checks for this rule through the `Dead code` and `Useless if` checkers..
- Code Prover does not use run-time checks to detect violations of this rule. Instead, Code Prover detects the violations at compile time.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Boolean Operations with Invariant Results

```
void func (unsigned int arg) {
 if (arg >= 0U) //Noncompliant
     arg  = 1U;
 if (arg < 0U) //Noncompliant
     arg = 1U;
}
```

An `unsigned int` variable is nonnegative. Both `if` conditions involving the variable are always true or always false and are therefore redundant.

Check Information

Group: Language Independent Issues
Category: Required

See Also

Topics
“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 0-1-3

A project shall not contain unused variables

Description

Rule Definition

A project shall not contain unused variables.

Polyspace Implementation

The checker flags local or global variables that are declared or defined but not used anywhere in the source files. This specification also applies to members of structures and classes.

Additional Message in Report

A project shall not contain unused variables.

Variable is never used or used only in unreachable code.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of Named Bit Field for Padding

```
#include <iostream>
struct S {
    unsigned char b1 : 3;
    unsigned char pad: 1;  //Noncompliant
    unsigned char b2 : 4;
};
void init(struct S S_obj)
{
    S_obj.b1 = 0;
    S_obj.b2 = 0;
}
```

In this example, the bit field `pad` is used for padding the structure. Therefore, the field is never read or written and causes a violation of this rule. To avoid the violation, use an unnamed field for padding.

```
struct S {
    unsigned char b1 : 3;
    unsigned char : 1;
    unsigned char b2 : 4;
};
```

Check Information

Group: Language Independent Issues
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2018a

MISRA C++:2008 Rule 0-1-5

A project shall not contain unused type declarations

Description

Rule Definition

A project shall not contain unused type declarations.

Rationale

If a type is declared but not used, when reviewing the code later, it is unclear if the type is redundant or left unused by mistake.

Unused types can indicate coding errors. For instance, you declared a enumerated data type for some specialized data but used an integer type for the data.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Unused `enum` Declaration

```
enum switchValue {low, medium, high}; //Noncompliant

void operate(int userInput) {
    switch(userInput) {
        case 0: // Turn on low setting
                 break;
        case 1: // Turn on medium setting
                 break;
        case 2: // Turn on high setting
                 break;
        default: // Return error
    }
}
```

In this example, the enumerated type `switchValue` is not used. Perhaps the intention was to use the type as `switch` input like this.

```
enum switchValue {low, medium, high}; //Compliant

void operate(switchValue userInput) {
    switch(userInput) {
        case low: // Turn on low setting
                   break;
        case medium: // Turn on medium setting
                     break;
        case high: // Turn on high setting
                   break;
        default: // Return error
```

```
    }
}
```

Check Information

Group: Language Independent Issues
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2018a

MISRA C++:2008 Rule 0-1-7

The value returned by a function having a non- void return type that is not an overloaded operator shall always be used

Description

Rule Definition

The value returned by a function having a non- void return type that is not an overloaded operator shall always be used.

Rationale

The unused return value might indicate a coding error or oversight.

Overloaded operators are excluded from this rule because their usage must emulate built-in operators which might not use their return value.

Polyspace Implementation

The checker flags functions with non-`void` return if the return value is not used or not explicitly cast to a `void` type.

The checker does not flag the functions `memcpy`, `memset`, `memmove`, `strcpy`, `strncpy`, `strcat`, `strncat` because these functions simply return a pointer to their first arguments.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Return Value Not Used

```
#include <iostream>
#include <new>

int assignMemory(int * ptr){
     int res = 1;
     ptr = new (std::nothrow) int;
     if(ptr==NULL) {
         res = 0;
     }
     return res;
}

void main() {
    int val;
    int status;

    assignMemory(&val);   //Noncompliant
    status = assignMemory(&val); //Compliant
```

```
    (void)assignMemory(&val); //Compliant

}
```

The first call to the function `assignMemory` is noncompliant because the return value is not used. The second and third calls use the return value. The return value from the second call is assigned to a local variable.

The return value from the third call is cast to `void`. Casting to `void` indicates deliberate non-use of the return value and cannot be a coding oversight.

Check Information

Group: Language Independent Issues
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 0-1-9

There shall be no dead code

Description

Rule Definition

There shall be no dead code.

Rationale

If an operation is reachable but removing the operation does not affect program behavior, the operation constitutes dead code. For instance, suppose that a variable is never read following a write operation. The write operation is redundant.

The presence of dead code can indicate an error in the program logic. Because a compiler can remove dead code, its presence can cause confusion for code reviewers.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Redundant Operations

```
#define ULIM 10000

int func(int arg) {
    int res;
    res = arg*arg + arg;
    if (res > ULIM)
        res = 0; //Noncompliant
    return arg;
}
```

In this example, the operations involving `res` are redundant because the function `func` returns its argument `arg`. All operations involving `res` can be removed without changing the effect of the function.

The checker flags the last write operation on `res` because the variable is never read after that point. The dead code can indicate an unintended coding error. For instance, you intended to return the value of `res` instead of `arg`.

Check Information

Group: Language Independent Issues
Category: Required

See Also

Topics
“Check for Coding Standard Violations”

Introduced in R2016b

MISRA C++:2008 Rule 0-1-10

Every defined function shall be called at least once

Description

Rule Definition

Every defined function shall be called at least once.

Rationale

If a function with a definition is not called, it might indicate a serious coding error. For instance, the function call is unreachable or a different function is called unintentionally.

Polyspace Implementation

The checker detects situations where a static function is defined but not called at all in its translation unit.

Additional Message in Report

Every defined function shall be called at least once. The static function *`funcName`* is not called.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Uncalled Static Function

```
static void func1() {
}

static void func2() { //Noncompliant
}

void func3();

int main() {
    func1();
    return 0;
}
```

The `static` function `func2` is defined but not called.

The function `func3` is not called either, however, it is only declared and not defined. The absence of a call to `func3` does not violate the rule.

Check Information

Group: Language Independent Issues

Category: Required

See Also

Topics
“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 0-1-11

There shall be no unused parameters (named or unnamed) in nonvirtual functions

Description

Rule Definition

There shall be no unused parameters (named or unnamed) in nonvirtual functions.

Rationale

Unused parameters often indicate later design changes. You perhaps removed all uses of a specific parameter but forgot to remove the parameter from the parameter list.

Unused parameters constitute an unnecessary overhead. You can also inadvertently call the function with a different number of arguments causing a parameter mismatch.

Polyspace Implementation

The checker flags a function that has unused named parameters unless the function body is empty.

Additional Message in Report

Function *`funcName`* has unused parameters.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Unused Parameters

```
typedef int (*callbackFn) (int a, int b);

int callback_1 (int a, int b) { //Compliant
    return a+b;
}

int callback_2 (int a, int b) { //Noncompliant
    return a;
}

int callback_3 (int, int b) { //Compliant - flagged by Polyspace
    return b;
}

int getCallbackNumber();
int getInput();

void main() {
    callbackFn ptrFn;
    int n = getCallbackNumber();
```

```
    int x = getInput(), y = getInput();
    switch(n) {
        case 0: ptrFn = &callback_1; break;
        case 1: ptrFn = &callback_2; break;
        default: ptrFn = &callback_3; break;
    }

    (*ptrFn)(x,y);
}
```

In this example, the three functions `callback_1`, `callback_2` and `callback_3` are used as callback functions. One of the three functions is called via a function pointer depending on a value obtained at run time.

- Function `callback_1` uses all its parameters and does not violate the rule.
- Function `callback_2` does not use its parameter `a` and violates this rule.
- Function `callback_3` also does not use its first parameter but it does not violate the rule because the parameter is unnamed. However, Polyspace flags the unused parameter as a rule violation. If you see a violation of this kind, justify the violation with comments. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Check Information

Group: Language Independent Issues
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2016b

MISRA C++:2008 Rule 0-1-12

There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it

Description

Rule Definition

There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.

Rationale

Unused parameters often indicate later design changes. You perhaps removed all uses of a specific parameter but forgot to remove the parameter from the parameter list.

Unused parameters constitute an unnecessary overhead. You can also inadvertently call the function with a different number of arguments causing a parameter mismatch.

Polyspace Implementation

For each virtual function, the checker looks at all overrides of the function. If an override has a named parameter that is not used, the checker shows a violation on the original virtual function and lists the override as a supporting event.

Note that Polyspace checks for unused parameters in virtual functions within single translation units. For instance, if a base class contains a virtual method with an unused parameter but the derived class implementation of the method uses that parameter, the rule is not violated. However, if the base class and derived class are defined in different files, the checker, which operates file by file, flags a violation of this rule on the base class.

The checker does not flag unused parameters in functions with empty bodies.

Additional Message in Report

There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.

Function *`funcName`* has unused parameters.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Unused Parameter in Virtual Function

```
class base {
    public:
       virtual void assignVal (int arg1, int arg2) = 0; //Noncompliant
```

```
       virtual void assignAnotherVal (int arg1, int arg2) = 0;
};

class derived1: public base {
    public:
       virtual void assignVal (int arg1, int arg2) {
           arg1 = 0;
       }
       virtual void assignAnotherVal (int arg1, int arg2) {
           arg1 = 1;
       }
};

class derived2: public base {
    public:
       virtual void assignVal (int arg1, int arg2) {
           arg1 = 0;
       }
       virtual void assignAnotherVal (int arg1, int arg2) {
           arg2 = 1;
       }
};
```

In this example, the second parameter of the virtual method `assignVal` is not used in any of the derived class implementations of the method.

On the other hand, the implementation of the virtual method `assignAnotherVal` in derived class `derived1` uses the first parameter of the method. The implementation in `derived2` uses the second parameter. Both parameters of `assignAnotherVal` are used and therefore the virtual method does not violate the rule.

Check Information

Group: Language Independent Issues
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2016b

MISRA C++:2008 Rule 0-2-1

An object shall not be assigned to an overlapping object

Description

Rule Definition

An object shall not be assigned to an overlapping object.

Rationale

When you assign an object to another object with overlapping memory, the behavior is undefined.

The exceptions are:

- You assign an object to another object with exactly overlapping memory and compatible type.
- You copy one object to another with `memmove`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Assignment of Union Members

```
void func (void) {
    union {
        short i;
        int j;
    } a = {0}, b = {1};

    a.j = a.i;   //Noncompliant
    a = b;       //Compliant
}
```

In this example, the rule is violated when `a.i` is assigned to `a.j` because the two variables have overlapping regions of memory.

Check Information

Group: Language Independent Issues
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2016b

MISRA C++:2008 Rule 1-0-1

All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1"

Description

Rule Definition

All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".

Polyspace Implementation

The checker reports compilation errors as detected by a compiler that strictly adheres to the C++03 Standard (ISO/IEC 14882:2003).

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Additional Message in Report

The message has two parts:

- Rule statement:

  All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".
- Compilation error message such as:

  Expected a ;

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: General
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 2-3-1

Trigraphs shall not be used

Description

Rule Definition

Trigraphs shall not be used.

Rationale

You denote trigraphs with two question marks followed by a specific third character (for instance,`'??-'` represents a `'~'` (tilde) character and `'??)'` represents a `']'`). These trigraphs can cause accidental confusion with other uses of two question marks.

For instance, the string

```
"(Date should be in the form ??-??-??)"
```

is transformed to

```
"(Date should be in the form ~~]"
```

but this transformation might not be intended.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 2-5-1

Digraphs should not be used

Description

Rule Definition

Digraphs should not be used.

Rationale

Digraphs are a sequence of two characters that are supposed to be treated as a single character. The checker flags use of these digraphs:

- `<%`, indicating `[`
- `%>`, indicating `]`
- `<:`, indicating `{`
- `:>`, indicating `}`
- `%:`, indicating `#`
- `%:%:`

When developing or reviewing code with digraphs, the developer or reviewer can incorrectly consider the digraph as a sequence of separate characters.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Lexical Conventions
Category: Advisory

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 2-7-1

The character sequence /* shall not be used within a C-style comment

Description

Rule Definition

The character sequence / shall not be used within a C-style comment.*

Rationale

If your code contains a `/*` in a `/* */` comment, it typically means that you have inadvertently commented out code. See the example that follows.

Polyspace Implementation

You cannot justify a violation of this rule using source code annotations.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of /* in /* */ Comment

```
void foo() {
    /* Initializer functions
     setup();
    /* Step functions */
}
```

In this example, the call to `setup()` is commented out because the ending `*/` is omitted, perhaps inadvertently. The checker flags this issue by highlighting the `/*` in the `/* */` comment.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 2-10-1

Different identifiers shall be typographically unambiguous

Description

Rule Definition

Different identifiers shall be typographically unambiguous.

Rationale

When you use identifiers that are typographically close, you can confuse between them.

The identifiers should not differ by:

- The interchange of a lowercase letter with its uppercase equivalent.
- The presence or absence of the underscore character.
- The interchange of the letter `O` and the digit `0`.
- The interchange of the letter `I` and the digit `1`.
- The interchange of the letter `I` and the letter `l`.
- The interchange of the letter `S` and the digit `5`.
- The interchange of the letter `Z` and the digit `2`.
- The interchange of the letter `n` and the letter `h`.
- The interchange of the letter `B` and the digit `8`.
- The interchange of the letters `rn` and the letter `m`.

Polyspace Implementation

The rule checker does not consider the fully qualified names of variables when checking this rule.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Typographically Ambiguous Identifiers

```
void func(void) {
    int id1_numval;
    int id1_num_val;  /* Non-compliant */

    int id2_numval;
    int id2_numVal;   /* Non-compliant */
```

```
    int id3_lvalue;
    int id3_Ivalue;   /* Non-compliant */

    int id4_xyz;
    int id4_xy2;      /* Non-compliant */

    int id5_zer0;
    int id5_zer0;     /* Non-compliant */

    int id6_rn;
    int id6_m;        /* Non-compliant */
}
```

In this example, the rule is violated when identifiers that can be confused for each other are used.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 2-10-2

Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope

Description

Rule Definition

Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Rationale

The rule flags situations where the same identifier name is used in two variable declarations, one in an outer scope and the other in an inner scope.

```
int var;
...
{
...
  int var;
...
}
```

All uses of the name in the inner scope refers to the variable declared in the inner scope. However, a developer or code reviewer can incorrectly assume that the usage refers to the variable declared in the outer scope.

Polyspace Implementation

The rule checker flags all cases of variable shadowing including when:

- The same identifier name is used in an outer and inner named namespace.
- The same name is used for a class data member and a variable outside the class.
- The same name is used for a method in a base and derived class.

To exclude these cases, switch to the more modern standard AUTOSAR C++14 and check for the rule `AUTOSAR C++14 Rule A2-10-1`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Local Variable Hiding Global Variable

```
int varInit = 1;

void doSomething(void);

void step(void) {
    int varInit = 0; //Noncompliant
    if(varInit)
```

```
       doSomething();
}
```

In this example, `varInit` defined in `func` hides the global variable `varInit`. The `if` condition refers to the local `varInit` and the block is unreachable, but you might expect otherwise.

Loop Index Hiding Variable Outside Loop

```
void runSomeCheck(int);

void checkMatrix(int dim1, int dim2) {
  for(int index = 0; index < dim1; index++) {
      for(int index = 0; index < dim2; index++) {
          runSomeCheck(index);
      }
  }
}
```

In this example, the variable `index` defined in the inner `for` loop hides the variable with the same name in the outer loop.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 2-10-3

A typedef name (including qualification, if any) shall be a unique identifier

Description

Rule Definition

A typedef name (including qualification, if any) shall be a unique identifier.

Rationale

The rule flags identifier declarations where the identifier name is the same as a previously declared typedef name. When you use identifiers that are identical, you can confuse between them.

Polyspace Implementation

The checker does not flag situations where the conflicting names occur in different namespaces.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Additional Message in Report

A typedef name (including qualification, if any) shall be a unique identifier.

Identifier *`typeName`* should not be reused.

Already used as typedef name (*`fileName lineNumber`*).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Typedef Name Conflicting with Other Identifiers

```
namespace NS1 {
    typedef int WIDTH;
}

namespace NS2 {
    float WIDTH; //Compliant
}

void f1() {
    typedef int TYPE;
}

void f2() {
    float TYPE; //Noncompliant
}
```

In this example, the declaration of `TYPE` in `f2()` conflicts with a typedef declaration in `f1()`.

The checker does not flag the redeclaration of `WIDTH` because the two declarations belong to different namespaces.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 2-10-4

A class, union or enum name (including qualification, if any) shall be a unique identifier

Description

Rule Definition

A class, union or enum name (including qualification, if any) shall be a unique identifier.

Rationale

The rule flags identifier declarations where the identifier name is the same as a previously declared class, union or typedef name. When you use identifiers that are identical, you can confuse between them.

Polyspace Implementation

The checker does not flag situations where the conflicting names occur in different namespaces.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Additional Message in Report

A class, union or enum name (including qualification, if any) shall be a unique identifier.

Identifier *`tagName`* should not be reused.

Already used as tag name (*`fileName lineNumber`*).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Typedef Name Conflicting with Other Identifiers

```
void f1() {
    class floatVar {};
}

void f2() {
    float floatVar; //Noncompliant
}
```

In this example, the declaration of `floatVar` in `f2()` conflicts with a class declaration in `f1()`.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Topics
“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 2-10-5

The identifier name of a non-member object or function with static storage duration should not be reused

Description

Rule Definition

The identifier name of a non-member object or function with static storage duration should not be reused.

Rationale

The rule flags situations where the name of an identifier with static storage duration is reused. The rule applies even if the identifiers belong to different namespaces because the reuse leaves the chance that you mistake one identifier for the other.

Polyspace Implementation

The rule checker flags redefined functions only when there is a declaration.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Additional Message in Report

The identifier name of a non-member object or function with static storage duration should not be reused.

Identifier *name* should not be reused.

Already used as static identifier with static storage duration (*fileName lineNumber*).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Reuse of Identifiers in Different Namespaces

```
namespace NS1 {
    static int WIDTH;
}

namespace NS2 {
    float WIDTH; //Noncompliant
}
```

In this example, the identifier name `WIDTH` is reused in the two namespaces `NS1` and `NS2`.

Check Information

Group: Lexical Conventions
Category: Advisory

See Also

Topics
“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 2-10-6

If an identifier refers to a type, it shall not also refer to an object or a function in the same scope

Description

Rule Definition

If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.

Rationale

For compatibility with C, in C++, you are allowed to use the same name for a type and an object or function. However, the name reuse can cause confusion during development or code review.

Polyspace Implementation

If the identifier is a function and the function is both declared and defined, then the violation is reported only once.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Reuse of Name for Type and Object

```
struct vector{
    int x;
    int y;
    int z;
}vector; //Noncompliant
```

In this example, the name `vector` is used both for the structured data type and for an object of that type.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 2-13-1

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used

Description

Rule Definition

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

Rationale

Escape sequences are certain special characters represented in string and character literals. They are written with a backslash (\) followed by a character.

The C++ Standard (ISO/IEC 14882:2003, Sec. 2.13.2) defines a list of escape sequences. See Escape Sequences. Use of escape sequences (backslash followed by character) outside that list leads to undefined behavior.

Additional Message in Report

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

`\`*`char`* is not an ISO/IEC escape sequence.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Incorrect Escape Sequences

```
void func () {
  const char a[2] = "\k"; \\Noncompliant
  const char b[2] = "\b"; \\Compliant
}
```

In this example, `\k` is not a recognized escape sequence.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 2-13-2

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used

Description

Rule Definition

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.

Rationale

Octal constants are denoted by a leading zero. A developer or code reviewer can mistake an octal constant as a decimal constant with a redundant leading zero.

Octal escape sequences beginning with \ can also cause confusion. Inadvertently introducing an 8 or 9 in the digit sequence after \ breaks the escape sequence and introduces a new digit. A developer or code reviewer can ignore this issue and continue to treat the escape sequence as one digit.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of Octal Constants and Octal Escape Sequences

```
void func(void) {
  int busData[6];

  busData[0] = 100;
  busData[1] = 108;
  busData[2] = 052;      //Noncompliant
  busData[3] = 071;      //Noncompliant
  busData[4] = '\109';   //Noncompliant
  busData[5] = '\100';   //Noncompliant

}
```

The checker flags all octal constants (other than zero) and all octal escape sequences (other than `\0`).

In this example:

- The octal escape sequence contains the digit 9, which is not an octal digit. This escape sequence has implementation-defined behavior.
- The octal escape sequence `\100` represents the number 64, but the rule checker forbids this use.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Topics
“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 2-13-3

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type

Description

Rule Definition

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

Rationale

The signedness of a constant is determined from:

- Value of the constant.
- Base of the constant: octal, decimal or hexadecimal.
- Size of the various types.
- Any suffixes used.

Unless you use a suffix `u` or `U`, another developer looking at your code cannot determine easily whether a constant is signed or unsigned.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 2-13-4

Literal suffixes shall be upper case

Description

Rule Definition

Literal suffixes shall be upper case.

Rationale

Literal constants can end with the letter `l` (el). Enforcing literal suffixes to be upper case removes potential confusion between the letter `l` and the digit `1`.

For consistency, use upper case constants for other suffixes such as `U` (unsigned) and `F` (float).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of Literal Constants with Lower Case Suffix

```
const int a = 0l; //Noncompliant
const int b = 0L; //Compliant
```

In this example, both `a` and `b` are assigned the same literal constant. However, from a quick glance, one can mistakenly assume that `a` is assigned the value `01` (octal one).

Check Information

Group: Lexical Conventions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 2-13-5

Narrow and wide string literals shall not be concatenated

Description

Rule Definition

Narrow and wide string literals shall not be concatenated.

Rationale

Narrow string literals are enclosed in double quotes without a prefix. Wide string literals are enclosed in double quotes with a prefix `L` outside the quotes. See string literals.

Concatenation of narrow and wide string literals can lead to undefined behavior.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Concatenation of Narrow and Wide String Literals

```
char array[] = "Hello" "World";
wchar_t w_array[] = L"Hello" L"World";
wchar_t mixed[] = "Hello" L"World"; //Noncompliant
```

In this example, in the initialization of the array `mixed`, the narrow string literal `"Hello"` is concatenated with the wide string literal `L"World"`.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 3-1-1

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule

Description

Rule Definition

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

Rationale

If a header file with variable or function definitions appears in multiple inclusion paths, the header file violates the One Definition Rule possibly leading to unpredictable behavior. For instance, a source file includes the header file `include.h` and another header file, which also includes `include.h`.

Polyspace Implementation

The rule checker flags variable and function definitions in header files.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Basic Concepts
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 3-1-2

Functions shall not be declared at block scope

Description

Rule Definition

Functions shall not be declared at block scope.

Rationale

It is a good practice to place all declarations at the namespace level.

Additionally, if you declare a function at block scope, it is often not clear if the statement is a function declaration or an object declaration with a call to the constructor.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Function Declarations at Block Scope

```
class A {
};

void b1() {
    void func(); //Noncompliant
    A a();   //Noncompliant
}
```

In this example, the declarations of `func` and `a` are in the block scope of `b1`.

The second function declaration can cause confusion because it is not clear if `a` is a function that returns an object of type `A` or `a` is itself an object of type `A`.

Check Information

Group: Basic Concepts
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 3-1-3

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization

Description

Rule Definition

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

Rationale

Though you can declare an incomplete array type and later complete the type, specifying the array size during the first declaration makes the subsequent array access less error-prone.

Additional Message in Report

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

Size of array *`arrayName`* should be explicitly stated.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Array Size Unspecified During Declaration

```
int array[10];
extern int array2[]; //Noncompliant
int array3[]= {0,1,2};
extern int array4[10];
```

In the declaration of `array2`, the array size is unspecified.

Check Information

Group: Basic Concepts
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 3-2-1

All declarations of an object or function shall have compatible types

Description

Rule Definition

All declarations of an object or function shall have compatible types.

Rationale

If the declarations of an object or function in two different translation units have incompatible types, the behavior is undefined.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Basic Concepts
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 3-2-2

The One Definition Rule shall not be violated

Description

Rule Definition

The One Definition Rule shall not be violated.

Rationale

Violations of the One Definition Rule leads to undefined behavior.

Polyspace Implementation

The checker flags situations where the same function or object has multiple definitions and the definitions differ by some token.

Additional Message in Report

The One Definition Rule shall not be violated.

Declaration of class *`className`* violates the One Definition Rule:

it conflicts with other declaration (*`fileName lineNumber`*).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Different Tokens in Same Type Definition

This example uses two files:

- `file1.cpp`:

```
struct S
{
   int x;
   int y;
};
```

- `file2.cpp`:

```
struct S
{
   int y;
   int x;
};
```

In this example, both `file1.cpp` and `file2.cpp` define the structure `S`. However, the definitions switch the order of the structure fields.

Check Information

Group: Basic Concepts
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 3-2-3

A type, object or function that is used in multiple translation units shall be declared in one and only one file

Description

Rule Definition

A type, object or function that is used in multiple translation units shall be declared in one and only one file.

Rationale

If you declare an identifier in a header file, you can include the header file in any translation unit where the identifier is defined or used. In this way, you ensure consistency between:

- The declaration and the definition.
- The declarations in different translation units.

The rule enforces the practice of declaring external objects or functions in header files.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Basic Concepts
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 3-2-4

An identifier with external linkage shall have exactly one definition

Description

Rule Definition

An identifier with external linkage shall have exactly one definition.

Rationale

If an identifier has multiple definitions or no definitions, it can lead to undefined behavior.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Multiple Definitions of Identifier

This example uses two files:

- `file1.cpp`:

  ```
  int x = 0;
  ```

- `file2.cpp`:

  ```
  int x = 1;
  ```

The same identifier `x` is defined in both files.

Check Information

Group: Basic Concepts
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 3-3-1

Objects or functions with external linkage shall be declared in a header file

Description

Rule Definition

Objects or functions with external linkage shall be declared in a header file.

Rationale

If you declare a function or object in a header file, it is clear that the function or object is meant to be accessed in multiple translation units. If you intend to access the function or object from a single translation unit, declare it `static` or in an unnamed namespace.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Declaration in Header File Missing

This example uses two files:

- `decls.h`:

  ```
  extern int x;
  ```

- `file.cpp`:

  ```
  #include "decls.h"

  int x = 0;
  int y = 0; //Noncompliant
  static int z = 0;
  ```

In this example, the variable `x` is declared in a header file but the variable `y` is not. The variable `z` is also not declared in a header file but it is declared with the `static` specifier and does not have external linkage.

Check Information

Group: Basic Concepts
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 3-3-2

If a function has internal linkage then all re-declarations shall include the static storage class specifier

Description

Rule Definition

If a function has internal linkage then all re-declarations shall include the static storage class specifier.

Rationale

If a function declaration has the `static` storage class specifier, it has internal linkage. Subsequent redeclarations of the function have internal linkage even without the `static` specifier.

However, if you do not specify the `static` keyword explicitly, it is not immediately clear from a declaration whether the function has internal linkage.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Missing `static` Specifier from Redeclaration

```
static void func1 ();
static void func2 ();

void func1() {}  //Noncompliant
static void func2() {}
```

In this example, the function `func1` is declared `static` but defined without the `static` specifier.

Check Information

Group: Basic Concepts
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 3-4-1

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility

Description

Rule Definition

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

Rationale

Defining variables with the minimum possible block scope reduces the possibility that they might later be accessed unintentionally.

For instance, if an object is meant to be accessed in one function only, declare the object local to the function.

Polyspace Implementation

The rule checker determines if an object is used in one block only. If the object is used in one block but defined outside the block, the checker raises a violation.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of Global Variable in Single Function

```
static int countReset; //Noncompliant

volatile int check;

void increaseCount() {
    int count = countReset;
    while(check%2) {
     count++;
    }
}
```

In this example, the variable `countReset` is declared global used in one function only. A compliant solution declares the variable local to the function to reduce its visibility.

Check Information

Group: Basic Concepts
Category: Required

See Also

Topics
“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 3-9-1

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations

Description

Rule Definition

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

Rationale

If a redeclaration is not token-for-token identical to the previous declaration, it is not clear from visual inspection which object or function is being redeclared.

Polyspace Implementation

The rule checker compares the current declaration with the last seen declaration.

Additional Message in Report

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

Variable *`varName`* is not compatible with previous declaration.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Identical Declarations That Do Not Match Token for Token

```
typedef int* intptr;

int* map;
extern intptr map; //Noncompliant

intptr table;
extern intptr table; //Compliant
```

In this example, the variable `map` is declared twice. The second declaration uses a `typedef` which resolves to the type of the first declaration. Because of the `typedef`, the second declaration is not token-for-token identical to the first.

Check Information

Group: Basic Concepts
Category: Required

See Also

Topics
“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 3-9-2

typedefs that indicate size and signedness should be used in place of the basic numerical types

Description

Rule Definition

typedefs that indicate size and signedness should be used in place of the basic numerical types.

Rationale

When the amount of memory being allocated is important, using specific-length types makes it clear how much storage is being reserved for each object.

Polyspace Implementation

The rule checker does not raise violations in templates that are not instantiated.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Direct Use of Basic Numerical Types

```
typedef unsigned int uint32_t;

unsigned int x = 0;       //Noncompliant
uint32_t y = 0;  //Compliant
```

In this example, the declaration of `x` is noncompliant because it uses the basic type `int` directly.

Check Information

Group: Basic Concepts
Category: Advisory

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 3-9-3

The underlying bit representations of floating-point values shall not be used

Description

Rule Definition

The underlying bit representations of floating-point values shall not be used.

Rationale

The underlying bit representations of floating point values vary across compilers. If you directly use the underlying representation of floating point values, your program is not portable across implementations.

Polyspace Implementation

The rule checker flags conversions from pointers to floating point types into pointers to integer types, and vice versa.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Using Underlying Representation of Floating-Point Values

```
float fabs2(float f) {
    unsigned int* ptr = reinterpret_cast <unsigned int*> (&f); //Noncompliant
    *(ptr + 3) &= 0x7f;
    return f;
}
```

In this example, the `reinterpret_cast` attempts to cast a floating-point value to an integer and access the underlying bit representation of the floating point value.

Check Information

Group: Basic Concepts
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 4-5-1

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator

Description

Rule Definition

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.

Rationale

Operators other than the ones mentioned in the rule do not produce meaningful results with `bool` operands. Use of `bool` operands with these operators can indicate programming errors. For instance, you intended to use the logical operator `||` but used the bitwise operator `|` instead.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Compliant and Noncompliant Uses of `bool` Operands

```
void boolOperations() {
    bool lhs = true;
    bool rhs = false;

    int res;

    if(lhs & rhs) {} //Noncompliant
    if(lhs < rhs) {} //Noncompliant
    if(~rhs) {}      //Noncompliant
    if(lhs ^ rhs) {} //Noncompliant
    if(lhs == rhs) {} //Compliant
    if(!rhs) {}       //Compliant
    res = lhs? -1:1;  //Compliant
}
```

In this example, `bool` operands do not violate the rule when used with the `==`, `!` and the `?` operators.

Check Information

Group: Standard Conversions
Category: Required

See Also

Topics
“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 4-5-2

Expressions with type enum shall not be used as operands to built- in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=

Description

Rule Definition

Expressions with type enum shall not be used as operands to built- in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Standard Conversions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 4-5-3

Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator N

Description

Rule Definition

Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator. N

Rationale

The C++03 Standard only requires that the characters `'0'` to `'9'` have consecutive values. Other characters do not have well-defined values. If you use these characters in operations other than the ones mentioned in the rule, you implicitly use their underlying values and might see unexpected results.

Additional Message in Report

Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator. N

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Compliant and Noncompliant Uses of Character Operands

```
void charManipulations (char ch) {

    char initChar = 'a'; //Compliant
    char finalChar = 'z'; //Compliant

    if(ch == initChar) {} //Compliant
    if( (ch >= initChar) && (ch <= finalChar)) {} //Noncompliant
    else if( (ch >= '0') && (ch <= '9') ) {} //Compliant by exception
}
```

In this example, character operands do not violate the rule when used with the = and == operators. Character operands can also be used with relational operators as long as the comparison is performed with the digits `'0'` to `'9'`.

Check Information

Group: Standard Conversions
Category: Required

See Also

Topics
“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 4-10-1

NULL shall not be used as an integer value

Description

Rule Definition

NULL shall not be used as an integer value.

Rationale

In C++, you can use the literals 0 and NULL as both an integer and a null pointer constant. However, use of 0 as a null pointer constant or NULL as an integer can cause developer confusion.

This rule restricts the use of NULL to null pointer constants. `MISRA C++:2008 Rule 4-10-2` restricts the use of the literal 0 to integers.

Polyspace Implementation

The checker flags assignment of NULL to an integer variable or binary operations involving NULL and an integer. Assignments can be direct or indirect such as passing NULL as integer argument to a function.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Compliant and Noncompliant Uses of `NULL`

```
#include <cstddef>

void checkInteger(int);
void checkPointer(int *);

void main() {
    checkInteger(NULL); //Noncompliant
    checkPointer(NULL); //Compliant
}
```

In this example, the use of `NULL` as argument to the `checkInteger` function is noncompliant because the function expects an `int` argument.

Check Information

Group: Standard Conversions
Category: Required

See Also

Topics
“Check for Coding Standard Violations”

Introduced in R2018a

MISRA C++:2008 Rule 4-10-2

Literal zero (0) shall not be used as the null-pointer-constant

Description

Rule Definition

Literal zero (0) shall not be used as the null-pointer-constant.

Rationale

In C++, you can use the literals 0 and NULL as both an integer and a null pointer constant. However, use of 0 as a null pointer constant or NULL as an integer can cause developer confusion.

This rule restricts the use of the literal 0 to integers. `MISRA C++:2008 Rule 4-10-1` restricts the use of NULL to null pointer constants.

Polyspace Implementation

The checker flags assignment of 0 to a pointer variable or binary operations involving 0 and a pointer. Assignments can be direct or indirect such as passing 0 as pointer argument to a function.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Compliant and Noncompliant Uses of Literal 0

```
#include <cstddef>

void checkInteger(int);
void checkPointer(int *);

void main() {
    checkInteger(0); //Compliant
    checkPointer(0); //Noncompliant
}
```

In this example, the use of 0 as argument to the `checkPointer` function is noncompliant because the function expects an `int *` argument.

Check Information

Group: Standard Conversions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2018a

MISRA C++:2008 Rule 5-0-1

The value of an expression shall be the same under any order of evaluation that the standard permits

Description

Rule Definition

The value of an expression shall be the same under any order of evaluation that the standard permits.

Rationale

If an expression results in different values depending on the order of evaluation, its value becomes implementation-defined.

Polyspace Implementation

An expression can have different values under the following conditions:

- The same variable is modified more than once in the expression, or is both read and written.
- The expression allows more than one order of evaluation.

Therefore, the rule checker forbids expressions where a variable is modified more than once and can cause different results under different orders of evaluation. The rule checker also detects cases where a volatile variable is read more than once in an expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Variable Modified More Than Once in Expression

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);         /* Compliant */
    COPY_ELEMENT (i++);       /* Non-compliant  */
}
```

In this example, the rule is violated by the statement `COPY_ELEMENT(i++)` because `i++` occurs twice and the order of evaluation of the two expressions is unspecified.

Variable Modified and Used in Multiple Function Arguments

```
void f (unsigned int param1, unsigned int param2) {}

void main () {
    unsigned int i=0;
```

```
    f ( i++, i );                  /* Non-compliant */
}
```

In this example, the rule is violated because it is unspecified whether the operation `i++` occurs before or after the second argument is passed to `f`. The call `f(i++,i)` can translate to either `f(0,0)` or `f(0,1)`.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-2

Limited dependence should be placed on C++ operator precedence rules in expressions

Description

Rule Definition

Limited dependence should be placed on C++ operator precedence rules in expressions.

Rationale

Use parentheses to clearly indicate the order of evaluation.

Depending on operator precedence can cause the following issues:

- If you or another code reviewer reviews the code, the intended order of evaluation is not immediately clear.
- It is possible that the result of the evaluation does not meet your expectations. For instance:
  - In the operation `*p++`, it is possible that you expect the dereferenced value to be incremented. However, the pointer `p` is incremented before the dereference.
  - In the operation `(x == y | z)`, it is possible that you expect `x` to be compared with `y | z`. However, the == operation happens before the `|` operation.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Evaluation Order Dependent on Operator Precedence Rules

```
#include <cstdio>

void showbits(unsigned int x) {
    for(int i = (sizeof(int) * 8) - 1; i >= 0; i--) {
       (x & 1u << i) ? putchar('1') : putchar('0'); // Noncompliant
    }
    printf("\n");
}
```

In this example, the checker flags the operation `x & 1u << i` because the statement relies on operator precedence rules for the `<<` operation to happen before the `&` operation. If this is the intended order, the operation can be rewritten as `x & (1u << i)`.

Check Information

Group: Expressions
Category: Advisory

See Also

Topics
“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-3

A cvalue expression shall not be implicitly converted to a different underlying type

Description

Rule Definition

A cvalue expression shall not be implicitly converted to a different underlying type.

Rationale

This rule ensures that the result of the expression does not overflow when converted to a different type.

Polyspace Implementation

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

The underlying data type of a cvalue expression is the widest of operand data types in the expression. For instance, if you add two variables, one of type `int8_t` (`typedef` for `char`) and another of type `int32_t` (`typedef` for `int`), the addition has underlying type `int32_t`. If you assign the sum to a variable of type `int8_t`, the rule is violated.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Implicit Conversion of Cvalue Expression

```
typedef char int8_t;
typedef signed int int32_t;

void func ( )
  {
    int32_t s32;
    int8_t s8;
    s32 = s8 + s8; //Noncompliant
    s32 = s32 + s8; //Compliant
  }
```

In this example, the rule is violated when two variables of type `int8_t` are added and the result is assigned to a variable of type `int32_t`. The underlying type of the addition does not take into account the integer promotion involved and is simply the widest of operand data types, in this case, `int8_t`.

The rule is not violated if one of the operands has type `int32_t` and the result is assigned to a variable of type `int32_t`. In this case, the underlying data type of the addition is the same as the type of the variable to which the result is assigned.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-4

An implicit integral conversion shall not change the signedness of the underlying type

Description

Rule Definition

An implicit integral conversion shall not change the signedness of the underlying type.

Rationale

Some conversions from signed to unsigned data types can lead to implementation-defined behavior. You can see unexpected results from the conversion.

Polyspace Implementation

The checker flags implicit conversions from a signed to an unsigned integer data type or vice versa.

The checker assumes that `ptrdiff_t` is a signed integer.

Additional Message in Report

An implicit integral conversion shall not change the signedness of the underlying type.

Implicit conversion of one of the binary + operands whose underlying types are *`typename_1`* and *`typename_2`*.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Implicit Conversions that Change Signedness

```
typedef char int8_t;
typedef unsigned char uint8_t;

void func()
  {
    int8_t s8;
    uint8_t u8;

    s8 = u8; //Noncompliant
    u8 = s8 + u8; //Noncompliant
    u8 = static_cast< uint8_t > ( s8 ) + u8; //Compliant
}
```

In this example, the rule is violated when a variable with a variable with signed data type is implicitly converted to a variable with unsigned data type or vice versa. If the conversion is explicit, as in the preceding example, the rule violation does not occur.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-5

There shall be no implicit floating-integral conversions

Description

Rule Definition

There shall be no implicit floating-integral conversions.

Rationale

If you convert from a floating point to an integer type, you lose information. Unless you explicitly cast from floating point to an integer type, it is not clear whether the loss of information is intended. Additionally, if the floating-point value cannot be represented in the integer type, the behavior is undefined.

Conversion from an integer to floating-point type can result in an inexact representation of the value. The error from conversion can accumulate over later operations and lead to unexpected results.

Polyspace Implementation

The checker flags implicit conversions between floating-point types (`float` and `double`) and integer types (`short`, `int`, etc.).

This rule takes precedence over 5-0-4 and 5-0-6 if they apply at the same time.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Conversion Between Floating Point and Integer Types

```
typedef signed int int32_t;
typedef float float32_t;

void func ( )
  {
    float32_t f32;
    int32_t   s32;
    s32 = f32;   //Noncompliant
    f32 = s32;   //Noncompliant
    f32 = static_cast< float32_t > ( s32 ); //Compliant
  }
```

In this example, the rule is violated when a floating-point type is *implicitly* converted to an integer type. The violation does not occur if the conversion is explicit.

Check Information

Group: Expressions

Category: Required

See Also

Topics
“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-6

An implicit integral or floating-point conversion shall not reduce the size of the underlying type

Description

Rule Definition

An implicit integral or floating-point conversion shall not reduce the size of the underlying type.

Rationale

A conversion that reduces the size of the underlying type can result in loss of information. Unless you explicitly cast to the narrower type, it is not clear whether the loss of information is intended.

Polyspace Implementation

The checker flags implicit conversions that reduce the size of a type.

If the conversion is to a narrower integer with a different sign, then rule 5-0-4 takes precedence over rule 5-0-6. Only rule 5-0-4 is shown.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Conversion That Reduces Size of Type

```
typedef signed short int16_t;
typedef signed int int32_t;

void func ( )
  {
    int16_t   s16;;
    int32_t   s32;
    s16 = s32;   //Noncompliant
    s16 = static_cast< int16_t > ( s32 ); //Compliant
  }
```

In this example, the rule is violated when a type is *implicitly* converted to a narrower type. The violation does not occur if the conversion is explicit.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-7

There shall be no explicit floating-integral conversions of a cvalue expression

Description

Rule Definition

There shall be no explicit floating-integral conversions of a cvalue expression.

Rationale

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation (the widest of operand data types in the expression). For instance, in this example, the result of an integer division is then cast to a floating-point type.

```
short num;
short den;
float res;
res= static_cast<float> (num/den);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a floating-point division because of the later cast.

Additional Message in Report

There shall be no explicit floating-integral conversions of a cvalue expression.

Complex expression of underlying type *`typeBeforeConversion`* may only be cast to narrower integer type of same signedness, however the destination type is *`typeAfterconversion`*.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Conversion of Division Result from Integer to Floating Point

```
void func() {
     short num;
     short den;
     short res_short;
     float res_float;

     res_float = static_cast<float> (num/den); //Noncompliant

     res_short = num/den;
     res_float = static_cast<float> (res_float); //Compliant

}
```

In this example, the first cast on the division result violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the expression is evaluated with an underlying type `float`.
- The second cast makes it clear that the expression is evaluated with the underlying type `short`. The result is then cast to the type `float`.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-8

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression

Description

Rule Definition

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Rationale

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation (the widest of operand data types in the expression). For instance, in this example, the sum of two `short` operands is cast to the wider type `int`.

```
short op1;
short op2;
int res;
res= static_cast<int> (op1 + op2);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a sum with the underlying type `int` because of the later cast.

Additional Message in Report

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Complex expression of underlying type *`typeBeforeConversion`* may only be cast to narrower integer type of same signedness, however the destination type is *`typeAfterconversion`*.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Conversion of Sum to Wider Integer Type

```
void func() {
     short op1;
     short op2;
     int res;

     res = static_cast<int> (op1 + op2); //Noncompliant
     res = static_cast<int> (op1) + op2; //Compliant
```

```
}
```

In this example, the first cast on the sum violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the sum is evaluated with an underlying type `int`.
- The second cast first converts one of the operands to `int` so that the sum is actually evaluated with the underlying type `int`.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-9

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression

Description

Rule Definition

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

Rationale

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation (the widest of operand data types in the expression).. For instance, in this example, the sum of two `unsigned int` operands is cast to the type `int`.

```
unsigned int op1;
unsigned int op2;
int res;
res= static_cast<int> (op1 + op2);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a sum with the underlying type `int` because of the later cast.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Conversion of Sum to Wider Integer Type

```
typedef int int32_t;
typedef unsigned int uint32_t;

void func() {
     uint32_t op1;
     uint32_t op2;
     int32_t res;

     res = static_cast<int32_t> (op1 + op2); //Noncompliant
     res = static_cast<int32_t> (op1) +
           static_cast<int32_t> (op2); //Compliant

}
```

In this example, the first cast on the sum violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the sum is evaluated with an underlying type `int32_t`.
- The second cast first converts each of the operands to `int32_t` so that the sum is actually evaluated with the underlying type `int32_t`.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-10

If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand

Description

Rule Definition

If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-11

The plain char type shall only be used for the storage and use of character values

Description

Rule Definition

The plain char type shall only be used for the storage and use of character values.

Polyspace Implementation

The checker raises a violation when a value of signed or unsigned integer type is implicitly converted to the plain char type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2015a

MISRA C++:2008 Rule 5-0-12

Signed char and unsigned char type shall only be used for the storage and use of numeric values

Description

Rule Definition

Signed char and unsigned char type shall only be used for the storage and use of numeric values.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2015a

MISRA C++:2008 Rule 5-0-13

The condition of an if-statement and the condition of an iteration- statement shall have type bool

Description

Rule Definition

The condition of an if-statement and the condition of an iteration- statement shall have type bool.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-14

The first operand of a conditional-operator shall have type bool

Description

Rule Definition

The first operand of a conditional-operator shall have type bool.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-15

Array indexing shall be the only form of pointer arithmetic

Description

Rule Definition

Array indexing shall be the only form of pointer arithmetic.

Polyspace Implementation

The checker flags:

- Arithmetic operations on all pointers, for instance `p+I`, `I+p` and `p-I`, where `p` is a pointer and `I` an integer..
- Array indexing on nonarray pointers.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-17

Subtraction between pointers shall only be applied to pointers that address elements of the same array

Description

Rule Definition

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Polyspace Implementation

Use Bug Finder for this checker. The rule checker performs the same checks as `Subtraction or comparison between pointers to different arrays`. Code Prover can fail to detect some violations.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-18

>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array

Description

Rule Definition

>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.

Polyspace Implementation

Use Bug Finder for this checker. The rule checker performs the same checks as `Subtraction or comparison between pointers to different arrays`. Code Prover can fail to detect some violations.

The checker ignores casts when showing the violation on relational operator use with pointers types.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-19

The declaration of objects shall contain no more than two levels of pointer indirection

Description

Rule Definition

The declaration of objects shall contain no more than two levels of pointer indirection.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-20

Non-constant operands to a binary bitwise operator shall have the same underlying type

Description

Rule Definition

Non-constant operands to a binary bitwise operator shall have the same underlying type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-0-21

Bitwise operators shall only be applied to operands of unsigned underlying type

Description

Rule Definition

Bitwise operators shall only be applied to operands of unsigned underlying type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-2-1

Each operand of a logical && or || shall be a postfix-expression

Description

Rule Definition

Each operand of a logical && or || shall be a postfix-expression.

Polyspace Implementation

During preprocessing, violations of this rule are detected on the expressions in `#if` directives.

The checker allows exceptions on associativity (a && b && c), (a || b || c).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-2-2

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast

Description

Rule Definition

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-2-3

Casts from a base class to a derived class should not be performed on polymorphic types

Description

Rule Definition

Casts from a base class to a derived class should not be performed on polymorphic types.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Advisory

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-2-4

C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used

Description

Rule Definition

C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-2-5

A cast shall not remove any const or volatile qualification from the type of a pointer or reference

Description

Rule Definition

A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-2-6

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type

Description

Rule Definition

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-2-7

An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly

Description

Rule Definition

An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.

Polyspace Implementation

The checker flags all pointer conversions including between a pointer to a `struct` object and a pointer to the first member of the same `struct` type.

Indirect conversions from a pointer to non-pointer type are not detected.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-2-8

An object with integer type or pointer to void type shall not be converted to an object with pointer type

Description

Rule Definition

An object with integer type or pointer to void type shall not be converted to an object with pointer type.

Polyspace Implementation

The checker allows an exception on zero constants.

Objects with pointer type include objects with pointer-to-function type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-2-9

A cast should not convert a pointer type to an integral type

Description

Rule Definition

A cast should not convert a pointer type to an integral type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Advisory

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-2-10

The increment (++) and decrement (--) operators should not be mixed with other operators in an expression

Description

Rule Definition

The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Advisory

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-2-11

The comma operator, && operator and the || operator shall not be overloaded

Description

Rule Definition

The comma operator, && operator and the || operator shall not be overloaded.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-2-12

An identifier with array type passed as a function argument shall not decay to a pointer

Description

Rule Definition

An identifier with array type passed as a function argument shall not decay to a pointer.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-3-1

Each operand of the ! operator, the logical && or the logical || operators shall have type bool

Description

Rule Definition

Each operand of the ! operator, the logical && or the logical || operators shall have type bool.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-3-2

The unary minus operator shall not be applied to an expression whose underlying type is unsigned

Description

Rule Definition

The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-3-3

The unary & operator shall not be overloaded

Description

Rule Definition

The unary & operator shall not be overloaded.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-3-4

Evaluation of the operand to the sizeof operator shall not contain side effects

Description

Rule Definition

Evaluation of the operand to the sizeof operator shall not contain side effects.

Polyspace Implementation

The checker does not show a warning on volatile accesses and function calls

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-8-1

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand

Description

Rule Definition

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-14-1

The right hand operand of a logical && or || operator shall not contain side effects

Description

Rule Definition

The right hand operand of a logical && or || operator shall not contain side effects.

Polyspace Implementation

The checker does not show a warning on volatile accesses and function calls.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-18-1

The comma operator shall not be used

Description

Rule Definition

The comma operator shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 5-19-1

Evaluation of constant unsigned integer expressions should not lead to wrap-around

Description

Rule Definition

Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-2-1

Assignment operators shall not be used in sub-expressions

Description

Rule Definition

Assignment operators shall not be used in sub-expressions.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-2-2

Floating-point expressions shall not be directly or indirectly tested for equality or inequality

Description

Rule Definition

Floating-point expressions shall not be directly or indirectly tested for equality or inequality.

Polyspace Implementation

The checker detects the use of == or != with floating-point variables or expressions. The checker does not detect indirectly testing of equality, for instance, using the <= operator.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-2-3

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white - space character

Description

Rule Definition

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white - space character.

Polyspace Implementation

The checker considers a null statement as a line where the first character excluding comments is a semicolon. The checker flags situations where:

- Comments appear before the semicolon.

  For instance:

  ```
  /* wait for pin */ ;
  ```

- Comments appear immediately after the semicolon without a white space in between.

  For instance:

  ```
  ;// wait for pin
  ```

The checker also shows a violation when a second statement appears on the same line following the null statement.

For instance:

```
; count++;
```

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-3-1

The statement forming the body of a switch, while, do while or for statement shall be a compound statement

Description

Rule Definition

The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Rationale

A compound statement is included in braces.

If a block of code associated with an iteration or selection statement is not contained in braces, you can make mistakes about the association. For example:

- You can wrongly associate a line of code with an iteration or selection statement because of its indentation.
- You can accidentally place a semicolon following the iteration or selection statement. Because of the semicolon, the line following the statement is no longer associated with the statement even though you intended otherwise.

This checker enforces the practice of adding braces following a selection or iteration statement even for a single line in the body. Later, when more lines are added, the developer adding them does not need to note the absence of braces and include them.

Polyspace Implementation

The checker flags `for` loops where the first token following a `for` statement is not a left brace, for instance:

```
for (i=init_val; i > 0; i--)
   if (arr[i] < 0)
      arr[i] = 0;
```

Similar checks are performed for `switch`, `for` and `do..while` statements.

The second line of the message on the **Result Details** pane indicates which statement is violating the rule. For instance, in the preceding example, the second line of the message states that the `for` loop is violating the rule.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics
“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-4-1

An if (condition) construct shall be followed by a compound statement The else keyword shall be followed by either a compound statement, or another if statement

Description

Rule Definition

An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

Additional Message in Report

An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-4-2

All if else if constructs shall be terminated with an else clause

Description

Rule Definition

All if ... else if constructs shall be terminated with an else clause.

Additional Message in Report

All if ... else if constructs shall be terminated with an else clause.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-4-3

A switch statement shall be a well-formed switch statement

Description

Rule Definition

A switch statement shall be a well-formed switch statement.

Polyspace Implementation

The checker flags these situations:

- A statement occurs between the `switch` statement and the first `case` statement.

  For instance:

  ```
  switch(ch) {
    int temp;
    case 1:
      break;
    default:
      break;
  }
  ```
- A label or a jump statement such as `goto` or `return` occurs in the `switch` block.
- A variable is declared in a `case` statement (outside any block).

  For instance:

  ```
  switch(ch) {
    case 1:
      int temp;
      break;
    default:
      break;
  }
  ```

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-4-4

A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement

Description

Rule Definition

A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-4-5

An unconditional throw or break statement shall terminate every non - empty switch-clause

Description

Rule Definition

An unconditional throw or break statement shall terminate every non - empty switch-clause.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-4-6

The final clause of a switch statement shall be the default-clause

Description

Rule Definition

The final clause of a switch statement shall be the default-clause.

Polyspace Implementation

The checker detects `switch` statements that do not have a final `default` clause.

The checker does not raise a violation if the `switch` variable is an `enum` with finite number of values and you have a `case` clause for each value. For instance:

```
enum Colours { RED, BLUE, GREEN } colour;

switch ( colour ) {
      case RED:
        break;
      case BLUE:
        break;
      case GREEN:
        break;
}
```

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-4-7

The condition of a switch statement shall not have bool type

Description

Rule Definition

The condition of a switch statement shall not have bool type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-4-8

Every switch statement shall have at least one case-clause

Description

Rule Definition

Every switch statement shall have at least one case-clause.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-5-1

A for loop shall contain a single loop-counter which shall not have floating type

Description

Rule Definition

A for loop shall contain a single loop-counter which shall not have floating type.

Polyspace Implementation

The checker flags these situations:

- The `for` loop index has a floating point type.
- More than one loop counter is incremented in the `for` loop increment statement.

  For instance:

  ```
  for(i=0, j=0; i<10 && j < 10;i++, j++) {}
  ```
- A loop counter is not incremented in the `for` loop increment statement.

  For instance:

  ```
  for(i=0; i<10;) {}
  ```

  Even if you increment the loop counter in the loop body, the checker still raises a violation. According to the MISRA C++ specifications, a loop counter is one that is initialized in or prior to the loop expression, acts as an operand to a relational operator in the loop expression and *is modified in the loop expression*. If the increment statement in the loop expression is missing, the checker cannot find the loop counter modification and considers as if a loop counter is not present.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-5-2

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=

Description

Rule Definition

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-5-3

The loop-counter shall not be modified within condition or statement

Description

Rule Definition

The loop-counter shall not be modified within condition or statement.

Rationale

The `for` loop has a specific syntax for modifying the loop counter. A code reviewer expects modification using that syntax. Modifying the loop counter elsewhere can make the code harder to review.

Polyspace Implementation

The checker flags modification of a `for` loop counter in the loop body or the loop condition (the condition that is checked to see if the loop must be terminated).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-5-4

The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop

Description

Rule Definition

The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-5-5

A loop-control-variable other than the loop-counter shall not be modified within condition or expression

Description

Rule Definition

A loop-control-variable other than the loop-counter shall not be modified within condition or expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-5-6

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool

Description

Rule Definition

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-6-1

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement

Description

Rule Definition

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-6-2

The goto statement shall jump to a label declared later in the same function body

Description

Rule Definition

The goto statement shall jump to a label declared later in the same function body.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-6-3

The continue statement shall only be used within a well-formed for loop

Description

Rule Definition

The continue statement shall only be used within a well-formed for loop.

Polyspace Implementation

The checker flags the use of `continue` statements in:

- `for` loops that are not well-formed, that is, loops that violate rules 6-5-x.
- `while` loops.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-6-4

For any iteration statement there shall be no more than one break or goto statement used for loop termination

Description

Rule Definition

For any iteration statement there shall be no more than one break or goto statement used for loop termination.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 6-6-5

A function shall have a single point of exit at the end of the function

Description

Rule Definition

A function shall have a single point of exit at the end of the function.

Rationale

This rule requires that a `return` statement must occur as the last statement in the function body. Otherwise, the following issues can occur:

- Code following a `return` statement can be unintentionally omitted.
- If a function that modifies some of its arguments has early `return` statements, when reading the code, it is not immediately clear which modifications actually occur.

Polyspace Implementation

The checker flags these situations:

- A function has more than one `return` statement.
- A non-`void` function has one `return` statement only but the `return` statement is not the last statement in the function.

A `void` function need not have a `return` statement. If a `return` statement exists, it need not be the last statement in the function.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 7-1-1

A variable which is not modified shall be const qualified

Description

Rule Definition

A variable which is not modified shall be const qualified.

Polyspace Implementation

The checker flags function parameters or local variables that are not const-qualified but never modified in the function body. Function parameters of integer, float, enum and boolean types are not flagged.

If a variable is passed to another function by reference or pointers, the checker assumes that the variable can be modified. These variables are not flagged.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarations
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2018a

MISRA C++:2008 Rule 7-1-2

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified

Description

Rule Definition

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Polyspace Implementation

The checker flags pointers where the underlying object is not const-qualified but never modified in the function body.

If a variable is passed to another function by reference or pointers, the checker assumes that the variable can be modified. Pointers that point to these variables are not flagged.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarations
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2018a

MISRA C++:2008 Rule 7-3-1

The global namespace shall only contain main, namespace declarations and extern "C" declarations

Description

Rule Definition

The global namespace shall only contain main, namespace declarations and extern "C" declarations.

Rationale

The rule makes sure that all names found at global scope are part of a namespace. Adhering to this rule avoids name clashes and ensures that developers do not reuse a variable name, resulting in compilation/linking errors, or shadow a variable name, resulting in possibly unexpected issues later.

Polyspace Implementation

Other than the `main` function, the checker flags all names used at global scope that are not part of a namespace.

The checker does not flag names at global scope if they are declared in `extern "C"` blocks (C code included within C++ code). However, if you use the option `Ignore link errors (-no-extern-c)`, these names are also flagged.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarations
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 7-3-2

The identifier main shall not be used for a function other than the global function main

Description

Rule Definition

The identifier main shall not be used for a function other than the global function main.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarations
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 7-3-3

There shall be no unnamed namespaces in header files

Description

Rule Definition

There shall be no unnamed namespaces in header files.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarations
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 7-3-4

using-directives shall not be used

Description

Rule Definition

using-directives shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarations
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 7-3-5

Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier

Description

Rule Definition

Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarations
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 7-3-6

using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files

Description

Rule Definition

using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarations
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 7-4-2

Assembler instructions shall only be introduced using the asm declaration

Description

Rule Definition

Assembler instructions shall only be introduced using the asm declaration.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarations
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 7-4-3

Assembly language shall be encapsulated and isolated

Description

Rule Definition

Assembly language shall be encapsulated and isolated.

Polyspace Implementation

The checker flags `asm` statements unless they are encapsulated in a function call.

For instance, the noncompliant `asm` statement below is in regular C code while the compliant `asm` statement is encapsulated in a call to the function `Delay`.

```
void Delay ( void )
  {
    asm( "NOP");//Compliant
  }
void fn (void)
  {
    DoSomething();
    Delay();// Assembler is encapsulated
    DoSomething();
    asm("NOP"); //Noncompliant
    DoSomething();
  }
```

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarations
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 7-5-1

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function

Description

Rule Definition

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarations
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 7-5-2

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist

Description

Rule Definition

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Polyspace Implementation

The checker flags situations where the address of a local variable is assigned to a pointer defined at global scope.

The checker does not raise violations of this rule if :

- A function returns the address of a local variable. This situation is covered by `MISRA C++:2008 Rule 7-5-1`.
- The address of a variable defined at block scope is assigned to a pointer that is defined with greater scope (but not global scope).

  For instance:

```
void foobar ( void )
 {
   char * ptr;
   {
     char var;
     ptr = &var;
   }
 }
```

  Only if the pointer is defined at global scope is the issue detected. For instance, the rule checker flags the issue here:

```
char * ptr;
void foobar ( void )
  {
      char var;
      ptr = &var;
  }
```

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarations
Category: Required

See Also

Topics
“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 7-5-3

A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference

Description

Rule Definition

A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarations
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 7-5-4

Functions should not call themselves, either directly or indirectly

Description

Rule Definition

Functions should not call themselves, either directly or indirectly.

Polyspace Implementation

The checker reports each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported.

You can calculate the total number of recursion cycles using the code complexity metric `Number of Recursions`. Note that unlike the checker, the metric also considers implicit calls, for instance, to compiler-generated constructors during object creation.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarations
Category: Advisory

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 8-0-1

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively

Description

Rule Definition

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 8-3-1

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments

Description

Rule Definition

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 8-4-1

Functions shall not be defined using the ellipsis notation

Description

Rule Definition

Functions shall not be defined using the ellipsis notation.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 8-4-2

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration

Description

Rule Definition

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.

Polyspace Implementation

The checker detects mismatch in parameter names between:

- A function declaration and the corresponding definition.
- Two declarations of a function, provided they occur in the same file.

  If the declarations occur in different files, the checker does not raise a violation for mismatch in parameter names. Redeclarations in different files are forbidden by `MISRA C++:2008 Rule 3-2-3`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 8-4-3

All exit paths from a function with non- void return type shall have an explicit return statement with an expression

Description

Rule Definition

All exit paths from a function with non- void return type shall have an explicit return statement with an expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 8-4-4

A function identifier shall either be used to call the function or it shall be preceded by &

Description

Rule Definition

A function identifier shall either be used to call the function or it shall be preceded by &.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 8-5-1

All variables shall have a defined value before they are used

Description

Rule Definition

All variables shall have a defined value before they are used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 8-5-2

Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures

Description

Rule Definition

Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 8-5-3

In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized

Description

Rule Definition

In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 9-3-1

const member functions shall not return non-const pointers or references to class-data

Description

Rule Definition

const member functions shall not return non-const pointers or references to class-data.

Polyspace Implementation

The checker flags a rule violation only if a `const` member function returns a non-`const` pointer or reference to a nonstatic data member. The rule does not apply to static data members.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Classes
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 9-3-2

Member functions shall not return non-const handles to class-data

Description

Rule Definition

Member functions shall not return non-const handles to class-data.

Polyspace Implementation

The checker flags a rule violation only if a member function returns a non-`const` pointer or reference to a nonstatic data member. The rule does not apply to static data members.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Classes
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 9-3-3

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const

Description

Rule Definition

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.

Polyspace Implementation

The checker flags member functions that are not declared static but do not access a data member of the class. Such a function can be potentially declared static.

The checker flags member functions that are not declared const but do not modify a data member of the class. Such a function can be potentially declared const.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Classes
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2018a

MISRA C++:2008 Rule 9-5-1

Unions shall not be used

Description

Rule Definition

Unions shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Classes
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 9-6-2

Bit-fields shall be either bool type or an explicitly unsigned or signed integral type

Description

Rule Definition

Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Classes
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 9-6-3

Bit-fields shall not have enum type

Description

Rule Definition

Bit-fields shall not have enum type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Classes
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 9-6-4

Named bit-fields with signed integer type shall have a length of more than one bit

Description

Rule Definition

Named bit-fields with signed integer type shall have a length of more than one bit.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Classes
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 10-1-1

Classes should not be derived from virtual bases

Description

Rule Definition

Classes should not be derived from virtual bases.

Rationale

The use of virtual bases can lead to many confusing behaviors.

For instance, in an inheritance hierarchy involving a virtual base, the most derived class calls the constructor of the virtual base. Intermediate calls to the virtual base constructor are ignored.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of Virtual Bases

```
class Base {};
class Intermediate: public virtual Base {}; //Noncompliant
class Final: public Intermediate {};
```

In this example, the rule checker raises a violation when the `Intermediate` class is derived from the class `Base` with the `virtual` keyword.

The following behavior can be a potential source of confusion. When you create an object of type `Final`, the constructor of `Final` directly calls the constructor of `Base`. Any call to the `Base` constructor from the `Intermediate` constructor are ignored. You might see unexpected results if you do not take into account this behavior.

Check Information

Group: Derived Classes
Category: Advisory

See Also

`MISRA C++:2008 Rule 10-1-2`

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 10-1-2

A base class shall only be declared virtual if it is used in a diamond hierarchy

Description

Rule Definition

A base class shall only be declared virtual if it is used in a diamond hierarchy.

Rationale

This rule is less restrictive than `MISRA C++:2008 Rule 10-1-1`. Rule 10-1-1 forbids the use of a virtual base anywhere in your code because a virtual base can lead to potentially confusing behavior.

Rule 10-1-2 allows the use of virtual bases in the one situation where they are useful, that is, as a common base class in diamond hierarchies.

For instance, the following diamond hierarchy violates rule 10-1-1 but not rule 10-1-2.

```
class Base {};
class Intermediate1: public virtual Base {};
class Intermediate2: public virtual Base {};
class Final: public Intermediate1, public Intermediate2 {};
```

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Derived Classes
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 10-1-3

An accessible base class shall not be both virtual and non-virtual in the same hierarchy

Description

Rule Definition

An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

Rationale

The checker flags situations where the same class is inherited as a virtual base class and a non-virtual base class in the same derived class. These situations defeat the purpose of virtual inheritance and causes multiple copies of the base class sub-object in the derived class object.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Base Class Both Virtual and Non-Virtual in Same Hierarchy

```
class Base {};
class Intermediate1: virtual public Base {};
class Intermediate2: virtual public Base {};
class Intermediate3: public Base {};
class Final: public Intermediate1, Intermediate2, Intermediate3 {}; //Noncompliant
```

In this example, the class `Base` is inherited in `Final` both as a virtual and non-virtual base class. The `Final` object contains at least two copies of a `Base` sub-object.

Check Information

Group: Derived Classes
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 10-2-1

All accessible entity names within a multiple inheritance hierarchy should be unique

Description

Rule Definition

All accessible entity names within a multiple inheritance hierarchy should be unique.

Polyspace Implementation

The checker flags data members from different classes with conflicting names if the same class derives from these classes. For instance:

```
class B1
  {
    public:
      int count;
      void foo ( );
  };
class B2
  {
    public:
      int count;
      void foo ( );
  };

class D : public B1, public B2
  {
    public:
      void Bar ( )
      {
        ++B1::count;
        B1::foo ( );
      }
  };
```

If the data member access in the derived class is ambiguous, the analysis reports this issue as a compilation error and not a coding rule violation. For instance, a compilation error occurs in the preceding example if the class `D` is rewritten as:

```
class D : public B1, public B2
  {
    public:
      void Bar ( )
      {
        ++count;      // Is that B1::count or B2::count?
        foo ( );      // Is that B1::foo() or B2::foo()?
      }
  };
```

The checker does not check for conflicts between entities of different kinds, for instance, member functions against data members.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Derived Classes
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 10-3-1

There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy

Description

Rule Definition

There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.

Rationale

The checker flags virtual member functions that have multiple definitions on the same path in an inheritance hierarchy. If a function is defined multiple times, it can be ambiguous which implementation is used in a given function call.

Polyspace Implementation

The checker also raises a violation if a base class member function is redefined in the derived class without the `virtual` keyword.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Virtual Function Redefined in Derived Class

```
class Base {
    public:
      virtual void foo() {
     }
};

class Intermediate1: public virtual Base {
    public:
      virtual void foo() { //Noncompliant
      }
};

class Intermediate2: public virtual Base {
    public:
       void bar() {
          foo();  // Calls Base::foo()
      }
};

class Final: public Intermediate1, public Intermediate2 {
};
```

```
void main() {
    Intermediate2 intermediate2Obj;
    intermediate2Obj.bar(); // Calls Base::foo()
    Final finalObj;
    finalObj.bar(); //Calls Intermediate1::foo()
                    //but you might expect Base::foo()
}
```

In this example, the `virtual` function `foo` is defined in the base class `Base` and also in the derived class `Intermediate1`.

A potential source of confusion can be the following. The class `Final` derives from `Intermediate1` and also derives from `Base` through another path using `Intermediate2`.

- When an `Intermediate2` object calls the function `bar` that calls the function `foo`, the implementation of `foo` in `Base` is called. An `Intermediate2` object does not know of the implementation in `Intermediate1`.
- However, when a `Final` object calls the same function `bar` that calls the function `foo`, the implementation of `foo` in `Intermediate1` is called because of dominance of the more derived class.

You might see unexpected results if you do not take this behavior into account.

To prevent this issue, declare a function as pure virtual in the base class. For instance, you can declare the class `Base` as follows:

```
class Base {
    public:
      virtual void foo()=0;
};

void Base::foo() {
      //You can still define Base::foo()
}
```

Check Information

Group: Derived Classes
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 10-3-2

Each overriding virtual function shall be declared with the virtual keyword

Description

Rule Definition

Each overriding virtual function shall be declared with the virtual keyword.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Derived Classes
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 10-3-3

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual

Description

Rule Definition

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Derived Classes
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 11-0-1

Member data in non- POD class types shall be private

Description

Rule Definition

Member data in non- POD class types shall be private.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Member Access Control
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 12-1-1

An object's dynamic type shall not be used from the body of its constructor or destructor

Description

Rule Definition

An object's dynamic type shall not be used from the body of its constructor or destructor.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Special Member Functions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 12-1-2

All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes

Description

Rule Definition

All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Special Member Functions
Category: Advisory

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 12-1-3

All constructors that are callable with a single argument of fundamental type shall be declared explicit

Description

Rule Definition

All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Special Member Functions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 12-8-1

A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member

Description

Rule Definition

A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Special Member Functions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 12-8-2

The copy assignment operator shall be declared protected or private in an abstract class

Description

Rule Definition

The copy assignment operator shall be declared protected or private in an abstract class.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Special Member Functions
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 14-5-2

A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter

Description

Rule Definition

A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Templates
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 14-5-3

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter

Description

Rule Definition

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Templates
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 14-6-1

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->

Description

Rule Definition

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Templates
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 14-6-2

The function chosen by overload resolution shall resolve to a function declared previously in the translation unit

Description

Rule Definition

The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Templates
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 14-7-3

All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template

Description

Rule Definition

All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Templates
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 14-8-1

Overloaded function templates shall not be explicitly specialized

Description

Rule Definition

Overloaded function templates shall not be explicitly specialized.

Polyspace Implementation

The checker first checks within file scope to find overloads. The checker later looks for call to a specialized template function later. As a result, the checker flags all specializations of overloaded templates even if overloading occurs after the call.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Templates
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 14-8-2

The viable function set for a function call should either contain no function specializations, or only contain function specializations

Description

Rule Definition

The viable function set for a function call should either contain no function specializations, or only contain function specializations.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Templates
Category: Advisory

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 15-0-2

An exception object should not have pointer type

Description

Rule Definition

An exception object should not have pointer type.

Polyspace Implementation

The checker raises a violation if a `throw` statement throws an exception of pointer type.

The checker does not raise a violation if a NULL pointer is thrown as exception. Throwing a NULL pointer is forbidden by `MISRA C++:2008 Rule 15-1-2`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling
Category: Advisory

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 15-0-3

Control shall not be transferred into a try or catch block using a goto or a switch statement

Description

Rule Definition

Control shall not be transferred into a try or catch block using a goto or a switch statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 15-1-2

NULL shall not be thrown explicitly

Description

Rule Definition

NULL shall not be thrown explicitly.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 15-1-3

An empty throw (throw;) shall only be used in the compound- statement of a catch handler

Description

Rule Definition

An empty throw (throw;) shall only be used in the compound- statement of a catch handler.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 15-3-2

There should be at least one exception handler to catch all otherwise unhandled exceptions

Description

Rule Definition

There should be at least one exception handler to catch all otherwise unhandled exceptions.

Polyspace Implementation

The checker shows a violation if there is no `try`/`catch` in the `main` function or the `catch` does not handle all exceptions (with ellipsis `...`). The rule is not checked if a `main` function does not exist.

The checker does not determine if an exception of an unhandled type actually propagates to `main`.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling
Category: Advisory

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 15-3-3

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases

Description

Rule Definition

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 15-3-5

A class type exception shall always be caught by reference

Description

Rule Definition

A class type exception shall always be caught by reference.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 15-3-6

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class

Description

Rule Definition

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 15-3-7

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last

Description

Rule Definition

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 15-4-1

If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids

Description

Rule Definition

If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 15-5-1

A class destructor shall not exit with an exception

Description

Rule Definition

A class destructor shall not exit with an exception.

Polyspace Implementation

The checker flags exceptions thrown in the body of the destructor. If the destructor calls another function, the checker does not detect if that function throws an exception.

The checker does not detect these situations:

- A `catch` statement does not catch exceptions of all types that are thrown.

  The checker considers the presence of a `catch` statement corresponding to a `try` block as indication that an exception is caught.
- `throw` statements inside `catch` blocks

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 15-5-2

Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s)

Description

Rule Definition

Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).

Polyspace Implementation

The checker flags situations where the data type of the exception thrown does not match the exception type listed in the function specification.

For instance:

```
void goo ( ) throw ( Exception )
  {
    throw 21; // Non-compliant - int is not listed
  }
```

The checker limits detection to `throw` statements that are in the body of the function. If the function calls another function, the checker does not detect if the called function throws an exception.

The checker does not detect `throw` statements inside `catch` blocks.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 15-5-3

The terminate() function shall not be called implicitly

Description

Rule Definition

The terminate() function shall not be called implicitly.

Polyspace Implementation

The checker flags these situations when the terminate() function can be called implicitly:

- An exception escapes uncaught. This also violates `MISRA C++:2008 Rule 15-3-2`. For instance:
  - Before an exception is caught, it escapes through another function that throws an uncaught exception. For instance, a catch statement or exception handler invokes a copy constructor that throws an uncaught exception.
  - A throw expression with no operand rethrows an uncaught exception.
- A class destructor throws an exception. This also violates `MISRA C++:2008 Rule 15-5-1`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2018a

MISRA C++:2008 Rule 16-0-1

#include directives in a file shall only be preceded by other preprocessor directives or comments

Description

Rule Definition

#include directives in a file shall only be preceded by other preprocessor directives or comments.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-0-2

Macros shall only be #define 'd or #undef 'd in the global namespace

Description

Rule Definition

Macros shall only be #define 'd or #undef 'd in the global namespace.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-0-3

#undef shall not be used

Description

Rule Definition

#undef shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-0-4

Function-like macros shall not be defined

Description

Rule Definition

Function-like macros shall not be defined.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-0-5

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives

Description

Rule Definition

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-0-6

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##

Description

Rule Definition

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-0-7

Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator

Description

Rule Definition

Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-0-8

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token

Description

Rule Definition

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-1-1

The defined preprocessor operator shall only be used in one of the two standard forms

Description

Rule Definition

The defined preprocessor operator shall only be used in one of the two standard forms.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-1-2

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related

Description

Rule Definition

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-2-1

The preprocessor shall only be used for file inclusion and include guards

Description

Rule Definition

The preprocessor shall only be used for file inclusion and include guards.

Polyspace Implementation

The checker flags `#ifdef` and `#define` statements in files that are not include files.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-2-2

C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers

Description

Rule Definition

C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.

Polyspace Implementation

The checker flags `#define` statements where the macros expand to something other than include guards, type qualifiers or storage class specifiers such as `static`, `inline`, `volatile`, `auto`, `register` and `const`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-2-3

Include guards shall be provided

Description

Rule Definition

Include guards shall be provided.

Polyspace Implementation

The checker raises a violation if a header file does not contain an include guard.

For instance, this code uses an include guard for the `#define` and `#include` statements and does not violate the rule:

```
// Contents of a header file
#ifndef FILE_H

#define FILE_H
#include "libFile.h"

#endif
```

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-2-4

The ', ", /* or // characters shall not occur in a header file name

Description

Rule Definition

The ', ", / or // characters shall not occur in a header file name.*

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-2-5

The \ character should not occur in a header file name

Description

Rule Definition

The \ character should not occur in a header file name.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Advisory

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-2-6

The #include directive shall be followed by either a <filename> or "filename" sequence

Description

Rule Definition

The #include directive shall be followed by either a <filename> or "filename" sequence.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-3-1

There shall be at most one occurrence of the # or ## operators in a single macro definition

Description

Rule Definition

There shall be at most one occurrence of the # or ## operators in a single macro definition.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-3-2

The # and ## operators should not be used

Description

Rule Definition

The # and ## operators should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Advisory

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 16-6-1

All uses of the #pragma directive shall be documented

Description

Rule Definition

All uses of the #pragma directive shall be documented.

Polyspace Implementation

To check this rule, you must list the pragmas that are allowed in source files by using the option `Allowed pragmas (-allowed-pragmas)`. If Polyspace finds a pragma not in the allowed pragma list, a violation is raised.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives
Category: Document

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2016b

MISRA C++:2008 Rule 17-0-1

Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined

Description

Rule Definition

Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Library Introduction
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 17-0-2

The names of standard library macros and objects shall not be reused

Description

Rule Definition

The names of standard library macros and objects shall not be reused.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Library Introduction
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 17-0-3

The names of standard library functions shall not be overridden

Description

Rule Definition

The names of standard library functions shall not be overridden.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Library Introduction
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2018a

MISRA C++:2008 Rule 17-0-5

The setjmp macro and the longjmp function shall not be used

Description

Rule Definition

The setjmp macro and the longjmp function shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Library Introduction
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 18-0-1

The C library shall not be used

Description

Rule Definition

The C library shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language Support Library
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 18-0-2

The library functions atof, atoi and atol from library <cstdlib> shall not be used

Description

Rule Definition

The library functions atof, atoi and atol from library <cstdlib> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language Support Library
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 18-0-3

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used

Description

Rule Definition

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language Support Library
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 18-0-4

The time handling functions of library <ctime> shall not be used

Description

Rule Definition

The time handling functions of library <ctime> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language Support Library
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 18-0-5

The unbounded functions of library <cstring> shall not be used

Description

Rule Definition

The unbounded functions of library <cstring> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language Support Library
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 18-2-1

The macro offsetof shall not be used

Description

Rule Definition

The macro offsetof shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language Support Library
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 18-4-1

Dynamic heap memory allocation shall not be used

Description

Rule Definition

Dynamic heap memory allocation shall not be used.

Rationale

Dynamic memory allocation uses heap memory, which can lead to issues such as memory leaks, data inconsistency, memory exhaustion, and nondeterministic behavior.

Polyspace Implementation

The checker flags uses of the `malloc`, `calloc`, `realloc` and `free` functions, and non-placement versions of the `new` and `delete` operator.

The checker also flags uses of the `alloca` function. Though memory leak cannot happen with the `alloca` function, other issues associated with dynamic memory allocation can still occur.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language Support Library
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 18-7-1

The signal handling facilities of <csignal> shall not be used

Description

Rule Definition

The signal handling facilities of <csignal> shall not be used.

Rationale

Signal handling functions such as `signal` contains undefined and implementation-specific behavior.

You have to be very careful when using `signal` to avoid these behaviors.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language Support Library
Category: Required

See Also

`Function called from signal handler not asynchronous-safe|Return from computational exception signal handler|Shared data access within signal handler|Signal call in multithreaded program`

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 19-3-1

The error indicator errno shall not be used

Description

Rule Definition

The error indicator errno shall not be used.

Rationale

Observing this rule encourages the good practice of not relying on `errno` to check error conditions.

Checking `errno` is not sufficient to guarantee absence of errors. Functions such as `fopen` might not set `errno` on error conditions. Often, you have to check the return value of such functions for error conditions.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `errno`

```
#include <cstdlib>
#include <cerrno>

void func (const char* str) {
    errno = 0;  // Noncompliant
    int i = atoi(str);
    if(errno != 0) { // Noncompliant
        //Handle Error
    }
}
```

The use of `errno` violates this rule. The function `atoi` is not required to set `errno` if the input string cannot be converted to an integer. Checking `errno` later does not safeguard against possible failures in conversion.

Check Information

Group: Diagnostic Library
Category: Required

See Also

`Misuse of errno|Misuse of errno in a signal handler`

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

MISRA C++:2008 Rule 27-0-1

The stream input/output library <cstdio> shall not be used

Description

Rule Definition

The stream input/output library <cstdio> shall not be used.

Rationale

Functions in `cstdio` such as `gets`, `fgetpos`, `fopen`, `ftell`, etc. have unspecified, undefined and implementation-defined behavior.

For instance:

- The `gets` function:

  ```
  char * gets ( char * buf );
  ```

  does not check if the number of characters provided at the standard input exceeds the buffer `buf`. The function can have unexpected behavior when the input exceeds the buffer.
- The `fopen` function has implementation-specific behavior related to whether it sets `errno` on errors or whether it accepts additional characters following the standard mode specifiers.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `gets`

```
#include <cstdio>

void func() {
 char array[10];
 gets(array);
}
```

The use of `gets` violates this rule.

Check Information

Group: Input/output Library
Category: Required

See Also

Topics

“Check for Coding Standard Violations”

Introduced in R2013b

CERT C Rules and Recommendations

Acknowledgement

CERT C: Rule PRE30-C

Do not create a universal character name through concatenation

Description

Rule Definition

Do not create a universal character name through concatenation.

Polyspace Implementation

This checker checks for **Universal character name from token concatenation**.

Examples

Universal character name from token concatenation

Issue

Universal character name from token concatenation occurs when two preprocessing tokens joined with a ## operator create a universal character name. A universal character name begins with `\u` or `\U` followed by hexadecimal digits. It represents a character not found in the basic character set.

For instance, you form the character `\u0401` by joining two tokens:

```
#define assign(uc1, uc2, val) uc1##uc2 = val
...
assign(\u04, 01, 4);
```

Risk

The C11 Standard (Sec. 5.1.1.2) states that if a universal character name is formed by token concatenation, the behavior is undefined.

Fix

Use the universal character name directly instead of producing it through token concatenation.

Example - Universal Character Name from Token Concatenation

```
#define assign(uc1, uc2, val) uc1##uc2 = val

int func(void) {
    int \u0401 = 0;
    assign(\u04, 01, 4);
    return \u0401;
}
```

In this example, the `assign` macro, when expanded, joins the two tokens `\u04` and `01` to form the universal character name `\u0401`.

Correction — Use Universal Character Name Directly

One possible correction is to use the universal character name `\u0401` directly. The correction redefines the `assign` macro so that it does not join tokens.

```
#define assign(ucn, val) ucn = val

int func(void) {
    int \u0401 = 0;
    assign(\u0401, 4);
    return \u0401;
}
```

Check Information

Group: Rule 01. Preprocessor (PRE)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

PRE30-C

Introduced in R2019a

CERT C: Rule PRE31-C

Avoid side effects in arguments to unsafe macros

Description

Rule Definition

Avoid side effects in arguments to unsafe macros.

Polyspace Implementation

This checker checks for **Side effect in arguments to unsafe macro**.

Examples

Side effect in arguments to unsafe macro

Issue

Side effect in arguments to unsafe macro occurs when you call an unsafe macro with an expression that has a side effect.

- *Unsafe macro*: When expanded, an unsafe macro evaluates its arguments multiple times or does not evaluate its argument at all.

  For instance, the `ABS` macro evaluates its argument `x` twice.

  ```
  #define ABS(x) (((x) < 0) ? -(x) : (x))
  ```
- *Side effect*: When evaluated, an expression with a side effect modifies at least one of the variables in the expression.

  For instance, `++n` modifies `n`, but `n+1` does not modify `n`.

  The checker does not consider side effects in nested macros. The checker also does not consider function calls or volatile variable access as side effects.

Risk

If you call an unsafe macro with an expression that has a side effect, the expression is evaluated multiple times or not evaluated at all. The side effect can occur multiple times or not occur at all, causing unexpected behavior.

For instance, in the call `MACRO(++n)`, you expect only one increment of the variable `n`. If `MACRO` is an unsafe macro, the increment happens more than once or does not happen at all.

The checker flags expressions with side effects in the `assert` macro because the `assert` macro is disabled in non-debug mode. To compile in non-debug mode, you define the `NDEBUG` macro during compilation. For instance, in GCC, you use the flag `-DNDEBUG`.

Fix

Evaluate the expression with a side effect in a separate statement, and then use the result as a macro argument.

For instance, instead of:

```
MACRO(++n);
```

perform the operation in two steps:

```
++n;
MACRO(n);
```

Alternatively, use an inline function instead of a macro. Pass the expression with side effect as argument to the inline function.

The checker considers modifications of a local variable defined only in the block scope of a macro body as a side effect. This defect cannot happen since the variable is visible only in the macro body. If you see a defect of this kind, ignore the defect.

Example - Macro Argument with Side Effects

```
#define ABS(x) (((x) < 0) ? -(x) : (x))

void func(int n) {
  /* Validate that n is within the desired range */
  int m = ABS(++n);

  /* ... */
}
```

In this example, the ABS macro evaluates its argument twice. The second evaluation can result in an unintended increment.

Correction — Separate Evaluation of Expression from Macro Usage

One possible correction is to first perform the increment, and then pass the result to the macro.

```
#define ABS(x) (((x) < 0) ? -(x) : (x))

void func(int n) {
  /* Validate that n is within the desired range */
  ++n;
  int m = ABS(n);

  /* ... */
}
```

Correction — Evaluate Expression in Inline Function

Another possible correction is to evaluate the expression in an inline function.

```
static inline int iabs(int x) {
  return (((x) < 0) ? -(x) : (x));
}

void func(int n) {
  /* Validate that n is within the desired range */

int m = iabs(++n);

  /* ... */
}
```

Check Information

Group: Rule 01. Preprocessor (PRE)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

PRE31-C

Introduced in R2019a

CERT C: Rule PRE32-C

Do not use preprocessor directives in invocations of function-like macros

Description

Rule Definition

Do not use preprocessor directives in invocations of function-like macros.

Polyspace Implementation

This checker checks for **Preprocessor directive in macro argument**.

Examples

Preprocessor directive in macro argument

Issue

Preprocessor directive in macro argument occurs when you use a preprocessor directive in the argument to a function-like macro or a function that might be implemented as a function-like macro.

For instance, a `#ifdef` statement occurs in the argument to a `memcpy` function. The `memcpy` function might be implemented as a macro.

```
memcpy(dest, src,
    #ifdef PLATFORM1
      12
    #else
      24
    #endif
  );
```

The checker flags similar usage in `printf` and `assert`, which can also be implemented as macros.

Risk

During preprocessing, a function-like macro call is replaced by the macro body and the parameters are replaced by the arguments to the macro call (argument substitution). Suppose a macro `min()` is defined as follows.

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

When you call `min(1,2)`, it is replaced by the body `((X) < (Y) ? (X) : (Y))`. `X` and `Y` are replaced by 1 and 2.

According to the C11 Standard (Sec. 6.10.3), if the list of arguments to a function-like macro itself has preprocessing directives, the argument substitution during preprocessing is undefined.

Fix

To ensure that the argument substitution happens in an unambiguous manner, use the preprocessor directives outside the function-like macro.

For instance, to execute `memcpy` with different arguments based on a `#ifdef` directive, call `memcpy` multiple times within the `#ifdef` directive branches.

```
#ifdef PLATFORM1
    memcpy(dest, src, 12);
#else
    memcpy(dest, src, 24);
#endif
```

Example - Directives in Function-Like Macros

```
#include <stdio.h>

#define print(A) printf(#A)

void func(void) {
    print(
#ifdef SW
          "Message 1"
#else
          "Message 2"
#endif
         );
}
```

In this example, the preprocessor directives `#ifdef` and `#endif` occur in the argument to the function-like macro `print()`.

Correction — Use Directives Outside Macro

One possible correction is to use the function-like macro multiple times in the branches of the `#ifdef` directive.

```
#include <stdio.h>

#define print(A) printf(#A)

void func(void) {
#ifdef SW
        print("Message 1");
#else
        print("Message 2");
#endif
}
```

Check Information

Group: Rule 01. Preprocessor (PRE)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

PRE32-C

Introduced in R2019a

CERT C: Rule DCL30-C

Declare objects with appropriate storage durations

Description

Rule Definition

Declare objects with appropriate storage durations.

Polyspace Implementation

This checker checks for **Pointer or reference to stack variable leaving scope**.

Examples

Pointer or reference to stack variable leaving scope

Issue

Pointer or reference to stack variable leaving scope occurs when a pointer or reference to a local variable leaves the scope of the variable. For instance:

- A function returns a pointer to a local variable.
- A function performs the assignment `globPtr = &locVar`. `globPtr` is a global pointer variable and `locVar` is a local variable.
- A function performs the assignment `*paramPtr = &locVar`. `paramPtr` is a function parameter that is, for instance, an `int**` pointer and `locVar` is a local `int` variable.
- A C++ method performs the assignment `memPtr = &locVar`. `memPtr` is a pointer data member of the class the method belongs to. `locVar` is a variable local to the method.

The defect also applies to memory allocated using the `alloca` function. The defect does not apply to static, local variables.

Risk

Local variables are allocated an address on the stack. Once the scope of a local variable ends, this address is available for reuse. Using this address to access the local variable value outside the variable scope can cause unexpected behavior.

If a pointer to a local variable leaves the scope of the variable, Polyspace Bug Finder highlights the defect. The defect appears even if you do not use the address stored in the pointer. For maintainable code, it is a good practice to not allow the pointer to leave the variable scope. Even if you do not use the address in the pointer now, someone else using your function can use the address, causing undefined behavior.

Fix

Do not allow a pointer or reference to a local variable to leave the variable scope.

Example - Pointer to Local Variable Returned from Function

```
void func2(int *ptr) {
    *ptr = 0;
```

```
}

int* func1(void) {
    int ret = 0;
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

In this example, `func1` returns a pointer to local variable `ret`.

In `main`, `ptr` points to the address of the local variable. When `ptr` is accessed in `func2`, the access is illegal because the scope of `ret` is limited to `func1`,

Check Information

Group: Rule 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL30-C

Introduced in R2019a

CERT C: Rule DCL31-C

Declare identifiers before using them

Description

Rule Definition

Declare identifiers before using them.

Polyspace Implementation

This checker checks for these issues:

- **Types not explicitly specified**.
- **Implicit function declaration**.

Examples

Types not explicitly specified

Issue

The rule checker flags situations where a function parameter or return type is not explicitly specified. To enable checking of this rule, use the value `c90` for the option `C standard version (-c-version)`.

Risk

In some circumstances, you can omit types from the C90 standard. In those cases, the `int` type is implicitly specified. However, the omission of an explicit type can lead to confusion. For example, in the declaration `extern void foo (char c, const k);`, the type of `k` is `const int`, but you might expect `const char`.

You might be using an implicit type in:

- Object declarations
- Parameter declarations
- Member declarations
- `typedef` declarations
- Function return types

Example - Implicit Types

```
static foo(int a);  /* Non compliant */
static void bar(void);      /* Compliant */
```

In this example, the rule is violated because the return type of `foo` is implicit.

Implicit function declaration

Issue

The issue occurs when you call a function before you declare or define it.

Risk

An implicit declaration occurs when you call a function before declaring or defining it. When you declare a function explicitly before calling it, the compiler can match the argument and return types with the parameter types in the declaration. If an implicit declaration occurs, the compiler makes assumptions about the argument and return types. For instance, it assumes a return type of `int`. The assumptions might not agree with what you expect and cause undesired type conversions.

Example - Function Not Declared Before Call

```
#include <math.h>

extern double power3 (double val, int exponent);
int getChoice(void);

double func() {
    double res;
    int ch = getChoice();
    if(ch == 0) {
        res = power(2.0, 10);    /* Non-compliant */
    }
    else if( ch==1) {
        res = power2(2.0, 10);   /* Non-compliant */
    }
    else {
        res = power3(2.0, 10);   /* Compliant */
        return res;
    }
}

double power2 (double val, int exponent) {
    return (pow(val, exponent));
}
```

In this example, the rule is violated when a function that is not declared is called in the code. Even if a function definition exists later in the code, the rule violation occurs.

The rule is not violated when the function is declared before it is called in the code. If the function definition exists in another file and is available only during the link phase, you can declare the function in one of the following ways:

- Declare the function with the `extern` keyword in the current file.
- Declare the function in a header file and include the header file in the current file.

Check Information

Group: Rule 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics
“Check for Coding Standard Violations”

External Websites
DCL31-C

Introduced in R2019a

CERT C: Rule DCL36-C

Do not declare an identifier with conflicting linkage classifications

Description

Rule Definition

Do not declare an identifier with conflicting linkage classifications.

Polyspace Implementation

This checker checks for **Inconsistent use of static and extern in object declarations**.

Examples

Inconsistent use of `static` and `extern` in object declarations

Issue

The issue occurs when you do not use the `static` storage class specifier consistently in all declarations of object and functions that have internal linkage.

The rule checker detects situations where:

- The same object is declared multiple times with different storage specifiers.
- The same function is declared and defined with different storage specifiers.

Risk

If you do not use the `static` specifier consistently in all declarations of objects with internal linkage, you might declare the same object with external and internal linkage.

In this situation, the linkage follows the earlier specification that is visible (C99 Standard, Section 6.2.2). For instance, if the earlier specification indicates internal linkage, the object has internal linkage even though the latter specification indicates external linkage. If you notice the latter specification alone, you might expect otherwise.

Example - Linkage Conflict Between Variable Declarations

```
static int foo = 0;
extern int foo;         /* Non-compliant */

extern int hhh;
static int hhh;         /* Non-compliant */
```

In this example, the first line defines `foo` with internal linkage. The first line is compliant because the example uses the `static` keyword. The second line does not use `static` in the declaration, so the declaration is noncompliant. By comparison, the third line declares `hhh` with an `extern` keyword creating external linkage. The fourth line declares `hhh` with internal linkage, but this declaration conflicts with the first declaration of `hhh`.

Correction — Consistent `static` and `extern` Use

One possible correction is to use `static` and `extern` consistently:

```
static int foo = 0;
static int foo;

extern int hhh;
extern int hhh;
```

Example - Linkage Conflict Between Function Declaration and Definition

```
static int fee(void);  /* Compliant - declaration: internal linkage */
int fee(void){         /* Non-compliant */
  return 1;
}

static int ggg(void);  /* Compliant - declaration: internal linkage */
extern int ggg(void){  /* Non-compliant */
  return 1 + x;
}
```

This example shows two internal linkage violations. Because `fee` and `ggg` have internal linkage, you must use a `static` class specifier to be compliant with MISRA.

Check Information

Group: Rule 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL36-C

Introduced in R2019a

CERT C: Rule DCL37-C

Do not declare or define a reserved identifier

Description

Rule Definition

Do not declare or define a reserved identifier.

Polyspace Implementation

This checker checks for these issues:

- **Defining and undefining reserved identifiers or macros**.
- **Declaring a reserved identifier or macro name**.

Examples

Defining and undefining reserved identifiers or macros

Issue

The issue occurs when you use `#define` and `#undef` on a reserved identifier or reserved macro name.

Risk

Reserved identifiers and reserved macro names are intended for use by the implementation. Removing or changing the meaning of a reserved macro can result in undefined behavior. This rule applies to the following:

- Identifiers or macro names beginning with an underscore
- Identifiers in file scope described in the C Standard Library (ISO/IEC 9899:1999, Section 7, "Library")
- Macro names described in the C Standard Library as being defined in a standard header (ISO/IEC 9899:1999, Section 7, "Library").

Example - Defining or Undefining Reserved Identifiers

```
#undef __LINE__                /* Non-compliant - begins with _ */
#define _Guard_H 1             /* Non-compliant - begins with _ */
#undef _ BUILTIN_sqrt         /* Non-compliant - implementation may
                                 * use _BUILTIN_sqrt for other purposes,
                                 * e.g. generating a sqrt instruction */
#define defined               /* Non-compliant - reserved identifier */
#define errno my_errno        /* Non-compliant - library identifier */
#define isneg(x) ( (x) < 0 )  /* Compliant - rule doesn't include
                                 * future library directions   */
```

Declaring a reserved identifier or macro name

Issue

The issue occurs when you declare a reserved identifier or macro name.

If you define a macro name that corresponds to a standard library macro, object, or function, Polyspace considers this a violation of the rule.

The rule considers tentative definitions as definitions.

Risk

The Standard allows implementations to treat reserved identifiers specially. If you reuse reserved identifiers, you can cause undefined behavior.

Check Information

Group: Rule 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL37-C

Introduced in R2019a

CERT C: Rule DCL38-C

Use the correct syntax when declaring a flexible array member

Description

Rule Definition

Use the correct syntax when declaring a flexible array member.

Polyspace Implementation

This checker checks for **Incorrect syntax of flexible array member size**.

Examples

Incorrect syntax of flexible array member size

Issue

Incorrect syntax of flexible array member size occurs when you do not use the standard C syntax to define a structure with a flexible array member.

Since C99, you can define a flexible array member with an unspecified size. For instance, `desc` is a flexible array member in this example:

```
struct record {
    size_t len;
    double desc[];
};
```

Prior to C99, you might have used compiler-specific methods to define flexible arrays. For instance, you used arrays of size one or zero:

```
struct record {
    size_t len;
    double desc[0];
};
```

This usage is not compliant with the C standards following C99.

Risk

If you define flexible array members by using size zero or one, your implementation is compiler-dependent. For compilers that do not recognize the syntax, an `int` array of size one has buffer for one `int` variable. If you try to write beyond this buffer, you can run into issues stemming from array access out of bounds.

If you use the standard C syntax to define a flexible array member, your implementation is portable across all compilers conforming with the standard.

Fix

To implement a flexible array member in a structure, define an array of unspecified size. The structure must have one member besides the array and the array must be the last member of the structure.

Example - Flexible Array Member Defined with Size One

```
#include <stdlib.h>

struct flexArrayStruct {
  int num;
  int data[1];
};

unsigned int max_size = 100;

void func(unsigned int array_size) {
  if(array_size<= 0 || array_size > max_size)
      exit(1);
  /* Space is allocated for the struct */
  struct flexArrayStruct *structP
    = (struct flexArrayStruct *)
     malloc(sizeof(struct flexArrayStruct)
          + sizeof(int) * (array_size - 1));
  if (structP == NULL) {
    /* Handle malloc failure */
    exit(2);
  }

  structP->num = array_size;

  /*
   * Access data[] as if it had been allocated
   * as data[array_size].
   */
  for (unsigned int i = 0; i < array_size; ++i) {
    structP->data[i] = 1;
  }

  free(structP);
}
```

In this example, the flexible array member `data` is defined with a size value of one. Compilers that do not recognize this syntax treat `data` as a size-one array. The statement `structP->data[i] = 1;` can write to `data` beyond the first array member and cause out of bounds array issues.

Correction — Use Standard C Syntax to Define Flexible Array

Define flexible array members with unspecified size.

```
#include <stdlib.h>

struct flexArrayStruct{
  int num;
  int data[];
};
```

```
unsigned int max_size = 100;

void func(unsigned int array_size) {
  if(array_size<=0 || array_size > max_size)
      exit(1);

  /* Allocate space for structure */
  struct flexArrayStruct *structP
    = (struct flexArrayStruct *)
    malloc(sizeof(struct flexArrayStruct)
         + sizeof(int) * array_size);

  if (structP == NULL) {
    /* Handle malloc failure */
    exit(2);
  }

  structP->num = array_size;

  /*
   * Access data[] as if it had been allocated
   * as data[array_size].
   */
  for (unsigned int i = 0; i < array_size; ++i) {
    structP->data[i] = 1;
  }

  free(structP);
}
```

Check Information

Group: Rule 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL38-C

Introduced in R2019a

CERT C: Rule DCL39-C

Avoid information leakage in structure padding

Description

Rule Definition

Avoid information leakage in structure padding.

Polyspace Implementation

This checker checks for **Information leak via structure padding**.

Examples

Information leak via structure padding

Issue

Information leak via structure padding occurs when you do not initialize the padding data of a structure or union before passing it across a trust boundary. A compiler adds padding bytes to the structure or union to ensure a proper memory alignment of its members. The bit-fields of the storage units can also have padding bits.

Information leak via structure padding raises a defect when:

- You call an untrusted function with structure or union pointer type argument containing uninitialized padding data.

  All external functions are considered untrusted.
- You copy or assign a structure or union containing uninitialized padding data to an untrusted object.

  All external structure or union objects, the output parameters of all externally linked functions, and the return pointer of all external functions are considered untrusted objects.

Risk

The padding bytes of the passed structure or union might contain sensitive information that an untrusted source can access.

Fix

- Prevent the addition of padding bytes for memory alignment by using the `pack` pragma or attribute supported by your compiler.
- Explicitly declare and initialize padding bytes as fields within the structure or union.
- Explicitly declare and initialize bit-fields corresponding to padding bits, even if you use the `pack` pragma or attribute supported by your compiler.

Example - Structure with Padding Bytes Passed to External Function

```
#include <stddef.h>
#include <stdlib.h>
```

```
#include <string.h>

typedef struct s_padding
{
 /* Padding bytes may be introduced between
 * 'char c' and 'int i'
 */
        char c;
    int i;

/*Padding bits may be introduced around the bit-fields
* even if you use "#pragma pack" (Windows) or
* __attribute__((__packed__)) (GNU)*/

    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;



/* External function */
extern void copy_object(void *out, void *in, size_t s);

void func(void *out_buffer)
{
/*Padding bytes not initialized*/

    S_Padding s = {'A', 10, 1, 3, {}};
/*Structure passed to external function*/

    copy_object((void *)out_buffer, (void *)&s, sizeof(s));
}

void main(void)
{
    S_Padding s1;
    func(&s1);
}
```

In this example, structure `s1` can have padding bytes between the `char c` and `int i` members. The bit-fields of the storage units of the structure can also contain padding bits. The content of the padding bytes and bits is accessible to an untrusted source when `s1` is passed to `func`.

Correction — Use pack Pragma to Prevent Padding Bytes

One possible correction in Microsoft Visual Studiois to use `#pragma pack()` to prevent padding bytes between the structure members. To prevent padding bits in the bit-fields of `s1`, explicitly declare and initialize the bit-fields even if you use `#pragma pack()`.

```
 #include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#define CHAR_BIT 8

#pragma pack(push, 1)
```

```
typedef struct s_padding
{
/*No Padding bytes when you use "#pragma pack" (Windows) or
* __attribute__((__packed__)) (GNU)*/
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
/* Padding bits explicitely declared */
    unsigned int bf_filler : sizeof(unsigned) * CHAR_BIT - 3;
    unsigned char buffer[20];
}

    S_Padding;

#pragma pack(pop)


/* External function */
extern void copy_object(void *out, void *in, size_t s);



void func(void *out_buffer)
{
    S_Padding s = {'A', 10, 1, 3, 0 /* padding bits */, {}};
    copy_object((void *)out_buffer, (void *)&s, sizeof(s));
}

void main(void)
{
    S_Padding s1;
    func(&s1);
}
```

Check Information

Group: Rule 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL39-C

Introduced in R2019a

CERT C: Rule DCL40-C

Do not create incompatible declarations of the same function or object

Description

Rule Definition

Do not create incompatible declarations of the same function or object.

Polyspace Implementation

This checker checks for **Declaration mismatch**.

Examples

Declaration mismatch

Issue

Declaration mismatch occurs when a function or variable declaration does not match other instances of the function or variable.

Risk

When a mismatch occurs between two variable declarations in different compilation units, a typical linker follows an algorithm to pick one declaration for the variable. If you expect a variable declaration that is different from the one chosen by the linker, you can see unexpected results when the variable is used.

A similar issue can occur with mismatch in function declarations.

Fix

The fix depends on the type of declaration mismatch. If both declarations indeed refer to the same object, use the same declaration. If the declarations refer to different objects, change the names of the one of the variables. If you change a variable name, remember to make the change in all places that use the variable.

Sometimes, declaration mismatches can occur because the declarations are affected by previous preprocessing directives. For instance, a declaration occurs in a macro, and the macro is defined on one inclusion path but undefined in another. These declaration mismatches can be tricky to debug. Identify the divergence between the two inclusion paths and fix the conflicting macro definitions.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Inconsistent Declarations in Two Files

file1.c

```
int foo(void) {
    return 1;
}
```

file2.c

```
double foo(void);

int bar(void) {
    return (int)foo();
}
```

In this example, *file1.c* declares `foo()` as returning an integer. In *file2.c*, `foo()` is declared as returning a double. This difference raises a defect on the second instance of `foo` in *file2*.

Correction — Align the Function Return Values

One possible correction is to change the function declarations so that they match. In this example, by changing the declaration of `foo` in *file2.c* to match *file1.c*, the defect is fixed.

file1.c

```
int foo(void) {
    return 1;
}
```

file2.c

```
int foo(void);

int bar(void) {
    return foo();
}
```

Example - Inconsistent Structure Alignment

test1.c	*test2.c*

`#include "square.h"`
`#include "circle.h"`
`struct aCircle circle;`
`struct aSquare square;`
 	
`int main(){`
`  square.side=1;`
`  circle.radius=1;`
`  return 0;`
`}`
	`#include "circle.h"`
	`#include "square.h"`
	`struct aCircle circle;`
	`struct aSquare square;`
	
	`int main(){`
	`  square.side=1;`
	`  circle.radius=1;`
	`  return 0;`
	`}`

circle.h	*square.h*

`#pragma pack(1)`
 	
`extern struct aCircle{`
`  int radius;`
`} circle;`
	`extern struct aSquare {`
	`  unsigned int side:1;`
	`} square;`

In this example, a declaration mismatch defect is raised on `square` in *square.h* because Polyspace infers that `square` in *square.h* does not have the same alignment as `square` in *test2.c*. This error occurs because the `#pragma pack(1)` statement in *circle.h* declares specific alignment. In *test2.c*, *circle.h* is included before *square.h*. Therefore, the `#pragma pack(1)` statement from *circle.h* is not reset to the default alignment after the `aCircle` structure. Because of this omission, *test2.c* infers that the `aSquare square` structure also has an alignment of `1` byte.

Correction — Close Packing Statements

One possible correction is to reset the structure alignment after the `aCircle` struct declaration. For the GNU or Microsoft Visual compilers, fix the defect by adding a `#pragma pack()` statement at the end of *circle.h*.

test1.c

```
#include "square.h"
#include "circle.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

test2.c

```
#include "circle.h"
#include "square.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

circle.h

```
#pragma pack(1)

extern struct aCircle{
    int radius;
} circle;

#pragma pack()
```

square.h

```
extern struct aSquare {
    unsigned int side:1;
} square;
```

Other compilers require different `#pragma pack` syntax. For your syntax, see the documentation for your compiler.

Correction — Use the `Ignore pragma pack directives` Option

One possible correction is to add the `Ignore pragma pack directives` option to your Bug Finder analysis. If you want the structure alignment to change for each structure, and you do not want to see this **Declaration mismatch** defect, use this correction.

1. On the Configuration pane, select the **Advanced Settings** pane.
2. In the **Other** box, enter `-ignore-pragma-pack`.
3. Rerun your analysis.

   The **Declaration mismatch** defect is resolved.

Check Information

Group: Rule 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL40-C

Introduced in R2019a

CERT C: Rule DCL41-C

Do not declare variables inside a switch statement before the first case label

Description

Rule Definition

Do not declare variables inside a switch statement before the first case label.

Polyspace Implementation

This checker checks for **switch statement not well-formed**.

Examples

`switch` statement not well-formed

Issue

The issue occurs when you define a variable in a `switch` block before the first `case` label.

Risk

In a `switch` block, control jumps to one of the `case` labels or a `default` label, depending on the control expression of the `switch` statement. If you define a variable before the first `case` label, the definition is ignored and later read operations on the variable in the `switch` block can lead to indeterminate values.

Check Information

Group: Rule 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics
“Check for Coding Standard Violations”

External Websites
DCL41-C

Introduced in R2019a

CERT C: Rule EXP30-C

Do not depend on the order of evaluation for side effects

Description

Rule Definition

Do not depend on the order of evaluation for side effects.

Polyspace Implementation

This checker checks for **Expression value depends on order of evaluation or of side effects**.

Examples

Expression value depends on order of evaluation or of side effects

Issue

The issue occurs when the value of an expression and its persistent side effects is not the same under all permitted evaluation orders.

An expression can have different values under the following conditions:

- The same variable is modified more than once in the expression, or is both read and written.
- The expression allows more than one order of evaluation.

Therefore, this rule forbids expressions where a variable is modified more than once and can cause different results under different orders of evaluation.

Risk

If an expression results in different values depending on the order of evaluation, its value becomes implementation-defined.

Example - Variable Modified More Than Once in Expression

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);         /* Compliant */
    COPY_ELEMENT (i++);       /* Noncompliant  */
}
```

In this example, the rule is violated by the statement `COPY_ELEMENT(i++)` because `i++` occurs twice and the order of evaluation of the two expressions is unspecified.

Example - Variable Modified and Used in Multiple Function Arguments

```
void f (unsigned int param1, unsigned int param2) {}
```

```
void main () {
    unsigned int i=0;
    f ( i++, i );                 /* Non-compliant */
}
```

In this example, the rule is violated because it is unspecified whether the operation `i++` occurs before or after the second argument is passed to `f`. The call `f(i++,i)` can translate to either `f(0,0)` or `f(0,1)`.

Check Information

Group: Rule 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP30-C

Introduced in R2019a

CERT C: Rule EXP32-C

Do not access a volatile object through a nonvolatile reference

Description

Rule Definition

Do not access a volatile object through a nonvolatile reference.

Polyspace Implementation

This checker checks for **Cast to pointer that removes volatile qualification**.

Examples

Cast to pointer that removes `volatile` qualification

Issue

Polyspace flags both implicit and explicit conversions that violate this rule.

Risk

This rule forbids casts from a pointer to a `volatile` object to a pointer that does not point to a `volatile` object. Such casts violate type qualification.

Example - Casts That Remove Qualifiers

```
void foo(void) {
    volatile unsigned short *pvi;    /* pointer to volatile  */
    unsigned short          *pi;

    pi  = (unsigned short *)  pvi;   /* Non-compliant */

}
```

In this example, the variable `pvi` has a `volatile` qualifier in its type. The rule is violated when the variable is cast to a type that does not have the `volatile` qualifier.

Check Information

Group: Rule 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP32-C

Introduced in R2019a

CERT C: Rule EXP33-C

Do not read uninitialized memory

Description

Rule Definition

Do not read uninitialized memory.

Polyspace Implementation

This checker checks for these issues:

- **Non-initialized pointer**.
- **Non-initialized variable**.

Examples

Non-initialized pointer

Issue

Non-initialized pointer occurs when a pointer is not assigned an address before dereference.

Risk

Unless a pointer is explicitly assigned an address, it points to an unpredictable location.

Fix

The fix depends on the root cause of the defect. For instance, you assigned an address to the pointer but the assignment is unreachable.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below. It is a good practice to initialize a pointer to NULL when declaring the pointer.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Non-initialized pointer error

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;
```

```
    if (prev == NULL)
      {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
      }

    *pi = j;
    /* Defect: Writing to uninitialized pointer */

    return pi;
}
```

If `prev` is not `NULL`, the pointer `pi` is not assigned an address. However, `pi` is dereferenced on every execution paths, irrespective of whether `prev` is `NULL` or not.

Correction — Initialize Pointer on Every Execution Path

One possible correction is to assign an address to `pi` when `prev` is not `NULL`.

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
       {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
       }
    /* Fix: Initialize pi in branches of if statement  */
    else
        pi = prev;


    *pi = j;

    return pi;
}
```

Non-initialized variable

Issue

Non-initialized variable occurs when a variable is not initialized before its value is read.

Risk

Unless a variable is explicitly initialized, the variable value is unpredictable. You cannot rely on the variable having a specific value.

Fix

The fix depends on the root cause of the defect. For instance, you assigned a value to the variable but the assignment is unreachable or you assigned a value to the variable in one of two branches of a conditional statement. Fix the unreachable code or missing assignment.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back

using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below. It is a good practice to initialize a variable at declaration.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Non-initialized variable error

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    int val;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
    /* Defect: val does not have a value if command is not 2 */
}
```

If `command` is not 2, the variable `val` is unassigned. In this case, the return value of function `get_sensor_value` is undetermined.

Correction — Initialize During Declaration

One possible correction is to initialize `val` during declaration so that the initialization is not bypassed on some execution paths.

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    /* Fix: Initialize val */
    int val=0;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
 }
```

`val` is assigned an initial value of 0. When `command` is not equal to 2, the function `get_sensor_value` returns this value.

Check Information

Group: Rule 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP33-C

Introduced in R2019a

CERT C: Rule EXP34-C

Do not dereference null pointers

Description

Rule Definition

Do not dereference null pointers.

Polyspace Implementation

This checker checks for **Null pointer**.

Examples

Null pointer

Issue

Null pointer occurs when you use a pointer with a value of `NULL` as if it points to a valid memory location.

Risk

Dereferencing a null pointer is undefined behavior. In most implementations, the dereference can cause your program to crash.

Fix

Check a pointer for `NULL` before dereference.

If the issue occurs despite an earlier check for `NULL`, look for intermediate events between the check and the subsequent dereference. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

Example - Null pointer error

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 int* p=NULL;

 *p=arr[0];
 /* Defect: Null pointer dereference */

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
```

```
  }

 return *p;
}
```

The pointer `p` is initialized with value of `NULL`. However, when the value `arr[0]` is written to `*p`, `p` is assumed to point to a valid memory location.

Correction — Assign Address to Null Pointer Before Dereference

One possible correction is to initialize `p` with a valid memory address before dereference.

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 /* Fix: Assign address to null pointer */
 int* p=&arr[0];

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

Check Information

Group: Rule 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP34-C

Introduced in R2019a

CERT C: Rule EXP35-C

Do not modify objects with temporary lifetime

Description

Rule Definition

Do not modify objects with temporary lifetime.

Polyspace Implementation

This checker checks for **Accessing object with temporary lifetime**.

Examples

Accessing object with temporary lifetime

Issue

Accessing object with temporary lifetime occurs when you attempt to read from or write to an object with temporary lifetime that is returned by a function call. In a structure or union returned by a function, and containing an array, the array members are temporary objects. The lifetime of temporary objects ends:

- When the full expression or full declarator containing the call ends, as defined in the C11 Standard.
- After the next sequence point, as defined in the C90 and C99 Standards. A sequence point is a point in the execution of a program where all previous evaluations are complete and no subsequent evaluation has started yet.

For C++ code, **Accessing object with temporary lifetime** raises a defect only when you write to an object with a temporary lifetime.

If the temporary lifetime object is returned by address, no defect is raised.

Risk

Modifying objects with temporary lifetime is undefined behavior and can cause abnormal program termination and portability issues.

Fix

Assign the object returned from the function call to a local variable. The content of the temporary lifetime object is copied to the variable. You can now modify it safely.

Example - Modifying Temporary Lifetime Object Returned by Function Call

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6
```

```
struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

/* func_temp() returns a struct value containing
* an array with a temporary lifetime.
*/
int func(void) {

/*Writing to temporary lifetime object is
 undefined behavior
 */
    return ++(func_temp().a[0]);
}

void main(void) {
    (void)func();
}
```

In this example, `func_temp()` returns by value a structure with an array member `a`. This member has temporary lifetime. Incrementing it is undefined behavior.

Correction — Assign Returned Value to Local Variable Before Writing

One possible correction is to assign the return of the call to `func_temp()` to a local variable. The content of the temporary object `a` is copied to the variable, which you can safely increment.

```
 #include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6

struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

int func(void) {

/* Assign object returned by function call to
 *local variable
 */
    struct S_Array s = func_temp();

/* Local variable can safely be
 *incremented
 */
    ++(s.a[0]);
```

```
    return s.a[0];
}

void main(void) {
    (void)func();
}
```

Check Information

Group: Rule 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP35-C

Introduced in R2019a

CERT C: Rule EXP36-C

Do not cast pointers into more strictly aligned pointer types

Description

Rule Definition

Do not cast pointers into more strictly aligned pointer types.

Polyspace Implementation

This checker checks for **Wrong allocated object size for cast**.

Examples

Wrong allocated object size for cast

Issue

Wrong allocated object size for cast occurs during pointer conversion when the pointer's address is misaligned. If a pointer is converted to a different pointer type, the size of the allocated memory must be a multiple of the size of the destination pointer.

Risk

Dereferencing a misaligned pointer has undefined behavior and can cause your program to crash.

Fix

Suppose you convert a pointer `ptr1` to `ptr2`. If `ptr1` points to a buffer of `N` bytes and `ptr2` is a *`type`* * pointer where `sizeof(`*`type`*`)` is `n` bytes, make sure that `N` is an integer multiple of `n`.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Dynamic Allocation of Pointers

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(13);
    long *dest;

    dest = (long*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to a `long`*. The dynamically allocated memory of `ptr`, 13 bytes, is not a multiple of the size of `dest`, 4 bytes. This misalignment causes the **Wrong allocated object size for cast** defect.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the allocated memory to 12 instead of 13.

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(12);
    long *dest;

    dest = (long*)ptr;
}
```

Example - Static Allocation of Pointers

```
void static_non_align(void){
    char arr[13], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to an `int*` in line 6. `ptr` has a memory size of 13 bytes because the array `arr` has a size of 13 bytes. The size of `dest` is 4 bytes, which is not a multiple of 13. This misalignment causes the **Wrong allocated object size for cast** defect.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the size of the array `arr` to a multiple of 4.

```
void static_non_align(void){
    char arr[12], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr;
}
```

Example - Allocation with a Function

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(13);  //defect
    dest2 = (char*)my_alloc(13); //not a defect
}
```

In this example, the software raises a defect on the conversion of the pointer returned by `my_alloc(13)` to an `int*` in line 11. `my_alloc(13)` returns a pointer with a dynamically allocated size of 13 bytes. The size of `dest1` is 4 bytes, which is not a divisor of 13. This misalignment causes the **Wrong allocated object size for cast** defect. In line 12, the same function call, `my_alloc(13)`, does not call a defect for the conversion to `dest2` because the size of `char*`, 1 byte, a divisor of 13.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the argument for `my_alloc` to a multiple of 4.

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(12);
    dest2 = (char*)my_alloc(13);
}
```

Check Information

Group: Rule 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP36-C

Introduced in R2019a

CERT C: Rule EXP37-C

Call functions with the correct number and type of arguments

Description

Rule Definition

Call functions with the correct number and type of arguments.

Polyspace Implementation

This checker checks for these issues:

- **Bad file access mode or status**.
- **Unreliable cast of function pointer**.
- **Standard function call with incorrect arguments**.

Examples

Bad file access mode or status

Issue

Bad file access mode or status occurs when you use functions in the `fopen` or `open` group with invalid or incompatible file access modes, file creation flags, or file status flags as arguments. For instance, for the `open` function, examples of valid:

- Access modes include `O_RDONLY`, `O_WRONLY`, and `O_RDWR`
- File creation flags include `O_CREAT`, `O_EXCL`, `O_NOCTTY`, and `O_TRUNC`.
- File status flags include `O_APPEND`, `O_ASYNC`, `O_CLOEXEC`, `O_DIRECT`, `O_DIRECTORY`, `O_LARGEFILE`, `O_NOATIME`, `O_NOFOLLOW`, `O_NONBLOCK`, `O_NDELAY`, `O_SHLOCK`, `O_EXLOCK`, `O_FSYNC`, `O_SYNC` and so on.

The defect can occur in the following situations.

Situation	Risk	Fix
You pass an empty or invalid access mode to the `fopen` function. According to the ANSI C standard, the valid access modes for `fopen` are: • `r`,`r+` • `w`,`w+` • `a`,`a+` • `rb`, `wb`, `ab` • `r+b`, `w+b`, `a+b` • `rb+`, `wb+`, `ab+`	`fopen` has undefined behavior for invalid access modes. Some implementations allow extension of the access mode such as: • GNU: `rb+cmxe,ccs=utf` • Visual C++: `a+t`, where `t` specifies a text mode. However, your access mode string must begin with one of the valid sequences.	Pass a valid access mode to `fopen`.
You pass the status flag `O_APPEND` to the `open` function without combining it with either `O_WRONLY` or `O_RDWR`.	`O_APPEND` indicates that you intend to add new content at the end of a file. However, without `O_WRONLY` or `O_RDWR`, you cannot write to the file. The `open` function does not return -1 for this logical error.	Pass either `O_APPEND|O_WRONLY` or `O_APPEND|O_RDWR` as access mode.
You pass the status flags `O_APPEND` and `O_TRUNC` together to the `open` function.	`O_APPEND` indicates that you intend to add new content at the end of a file. However, `O_TRUNC` indicates that you intend to truncate the file to zero. Therefore, the two modes cannot operate together. The `open` function does not return -1 for this logical error.	Depending on what you intend to do, pass one of the two modes.
You pass the status flag `O_ASYNC` to the `open` function.	On certain implementations, the mode `O_ASYNC` does not enable signal-driven I/O operations.	Use the `fcntl(pathname, F_SETFL, O_ASYNC);` instead.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Invalid Access Mode with `fopen`

```
#include <stdio.h>

void func(void) {
    FILE *file = fopen("data.txt", "rw");
    if(file!=NULL) {
        fputs("new data",file);
        fclose(file);
    }
}
```

In this example, the access mode `rw` is invalid. Because `r` indicates that you open the file for reading and `w` indicates that you create a new file for writing, the two access modes are incompatible.

Correction — Use Either `r` or `w` as Access Mode

One possible correction is to use the access mode corresponding to what you intend to do.

```
#include <stdio.h>

void func(void) {
    FILE *file = fopen("data.txt", "w");
    if(file!=NULL) {
        fputs("new data",file);
        fclose(file);
    }
}
```

Unreliable cast of function pointer

Issue

Unreliable cast of function pointer occurs when a function pointer is cast to another function pointer that has different argument or return type.

This defect applies only if the code language for the project is C.

Risk

If you cast a function pointer to another function pointer with different argument or return type and then use the latter function pointer to call a function, the behavior is undefined.

Fix

Avoid a cast between two function pointers with mismatch in argument or return types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Unreliable cast of function pointer error

```
#include <stdio.h>
#include <math.h>
#include <stdio.h>
#define PI 3.142
```

```
double Calculate_Sum(int (*fptr)(double))
{
    double  sum = 0.0;
    double  y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;

    fp = sin;
    sum = Calculate_Sum(fp);
    /* Defect: fp implicitly cast to int(*) (double) */

    printf("sum(sin): %f\n", sum);
    return 0;
}
```

The function pointer `fp` is declared as `double (*)(double)`. However in passing it to function `Calculate_Sum`, `fp` is implicitly cast to `int (*)(double)`.

Correction — Avoid Function Pointer Cast

One possible correction is to check that the function pointer in the definition of `Calculate_Sum` has the same argument and return type as `fp`. This step makes sure that `fp` is not implicitly cast to a different argument or return type.

```
#include <stdio.h>
#include <math.h>
#include <stdio.h>
# define PI 3.142

/*Fix: fptr has same argument and return type everywhere*/
double Calculate_Sum(double (*fptr)(double))
{
    double  sum = 0.0;
    double y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;
```

```
    fp = sin;
    sum = Calculate_Sum(fp);
    printf("sum(sin): %f\n", sum);

    return 0;
}
```

Standard function call with incorrect arguments

Issue

Standard function call with incorrect arguments occurs when the arguments to certain standard functions do not meet the requirements for their use in the functions.

For instance, the arguments to these functions can be invalid in the following ways.

Function Type	Situation	Risk	Fix
String manipulation functions such as `strlen` and `strcpy`	The pointer arguments do not point to a `NULL`-terminated string.	The behavior of the function is undefined.	Pass a `NULL`-terminated string to string manipulation functions.
File handling functions in `stdio.h` such as `fputc` and `fread`	The `FILE*` pointer argument can have the value `NULL`.	The behavior of the function is undefined.	Test the `FILE*` pointer for `NULL` before using it as function argument.
File handling functions in `unistd.h` such as `lseek` and `read`	The file descriptor argument can be -1.	The behavior of the function is undefined. Most implementations of the `open` function return a file descriptor value of -1. In addition, they set `errno` to indicate that an error has occurred when opening a file.	Test the return value of the `open` function for -1 before using it as argument for `read` or `lseek`. If the return value is -1, check the value of `errno` to see which error has occurred.
	The file descriptor argument represents a closed file descriptor.	The behavior of the function is undefined.	Close the file descriptor only after you have completely finished using it. Alternatively, reopen the file descriptor before using it as function argument.
Directory name generation functions such as `mkdtemp` and `mkstemps`	The last six characters of the string template are not `XXXXXX`.	The function replaces the last six characters with a string that makes the file name unique. If the last six characters are not `XXXXXX`, the function cannot generate a unique enough directory name.	Test if the last six characters of a string are `XXXXXX` before using the string as function argument.

Function Type	Situation	Risk	Fix
Functions related to environment variables such as `getenv` and `setenv`	The string argument is `""`.	The behavior is implementation-defined.	Test the string argument for `""` before using it as `getenv` or `setenv` argument.
	The string argument terminates with an equal sign, =. For instance, `"C="` instead of `"C"`.	The behavior is implementation-defined.	Do not terminate the string argument with =.
String handling functions such as `strtok` and `strstr`	• `strtok`: The delimiter argument is `""`. • `strstr`: The search string argument is `""`.	Some implementations do not handle these edge cases.	Test the string for `""` before using it as function argument.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - `NULL` Pointer Passed as `strnlen` Argument

```
#include <string.h>
#include <stdlib.h>

enum {
    SIZE10 = 10,
    SIZE20 = 20
};

int func() {
    char* s = NULL;
    return strnlen(s, SIZE20);
}
```

In this example, a `NULL` pointer is passed as `strnlen` argument instead of a `NULL`-terminated string.

Before running analysis on the code, specify a GNU compiler. See `Compiler (-compiler)`.

Correction — Pass `NULL`-terminated String

Pass a `NULL`-terminated string as the first argument of `strnlen`.

```
#include <string.h>
#include <stdlib.h>
```

```
enum {
    SIZE10 = 10,
    SIZE20 = 20
};

int func() {
    char* s = "";
    return strnlen(s, SIZE20);
}
```

Check Information

Group: Rule 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP37-C

Introduced in R2019a

CERT C: Rule EXP39-C

Do not access a variable through a pointer of an incompatible type

Description

Rule Definition

Do not access a variable through a pointer of an incompatible type.

Polyspace Implementation

This checker checks for **Cast to pointer pointing to object of different type**.

Examples

Cast to pointer pointing to object of different type

Issue

The issue occurs when you perform a cast between a pointer to an object type and a pointer to a different object type.

Risk

If a pointer to an object is cast into a pointer to a different object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.

Even if the conversion produces a pointer that is correctly aligned, the behavior can be undefined if the pointer is used to access an object.

Exception: You can convert a pointer to object type into a pointer to one of the following types:

- `char`
- `signed char`
- `unsigned char`

Example - Noncompliant: Cast to Pointer Pointing to Object of Wider Type

```
signed   char *p1;
unsigned int *p2;

void foo(void){
  p2 = ( unsigned int * ) p1;     /* Non-compliant */
}
```

In this example, `p1` can point to a `signed char` object. However, `p1` is cast to a pointer that points to an object of wider type, `unsigned int`.

Example - Noncompliant: Cast to Pointer Pointing to Object of Narrower Type

```
extern unsigned int read_value ( void );
extern void display ( unsigned int n );
```

```
void foo ( void ){
  unsigned int u = read_value ( );
  unsigned short *hi_p = ( unsigned short * ) &u;    /* Non-compliant  */
  *hi_p = 0;
  display ( u );
}
```

In this example, `u` is an `unsigned int` variable. `&u` is cast to a pointer that points to an object of narrower type, `unsigned short`.

On a big-endian machine, the statement `*hi_p = 0` attempts to clear the high bits of the memory location that `&u` points to. But, from the result of `display(u)`, you might find that the high bits have not been cleared.

Example - Compliant: Cast Adding a Type Qualifier

```
const short *p;
const volatile short *q;
void foo (void){
  q = ( const volatile short * ) p;  /* Compliant */
}
```

In this example, both `p` and `q` can point to `short` objects. The cast between them adds a `volatile` qualifier only and is therefore compliant.

Check Information

Group: Rule 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP39-C

Introduced in R2019a

CERT C: Rule EXP40-C

Do not modify constant objects

Description

Rule Definition

Do not modify constant objects.

Polyspace Implementation

This checker checks for **Writing to const qualified object**.

Examples

Writing to const qualified object

Issue

Writing to `const` qualified object occurs when you do one of the following:

- Use a `const`-qualified object as the destination of an assignment.
- Pass a `const`-qualified object to a function that modifies the argument.

For instance, the defect can occur in the following situations:

- You pass a `const`-qualified object as first argument of one of the following functions:
  - `mkstemp`
  - `mkostemp`
  - `mkostemps`
  - `mkdtemp`
- You pass a `const`-qualified object as the destination argument of one of the following functions:
  - `strcpy`
  - `strncpy`
  - `strcat`
  - `memset`
- You perform a write operation on a `const`-qualified object.

Risk

The risk depends upon the modifications made to the `const`-qualified object.

Situation	Risk
Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on.	These functions replace the last six characters of their first argument with a string. Therefore, they expect a modifiable `char` array as their first argument.
Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on.	These functions modify their destination argument. Therefore, they expect a modifiable `char` array as their destination argument.
Writing to the object	The `const` qualifier implies an agreement that the value of the object will not be modified. By writing to a `const`-qualified object, you break the agreement. The result of the operation is undefined.

Fix

The fix depends on the modification made to the `const`-qualified object.

Situation	Fix
Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on.	Pass a non-`const` object as first argument of the function.
Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on.	Pass a non-`const` object as destination argument of the function.
Writing to the object	Perform the write operation on a non-`const` object.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Writing to `const`-Qualified Object

```
#include <string.h>

const char* buffer = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

In this example, because `buffer` is `const`-qualified, `strchr(buffer,'X')` returns a `const`-qualified `char*` pointer. When this `char*` pointer is used as the destination argument of `strcpy`, a **Writing to const qualified object** error appears.

Correction — Copy `const`-Qualified Object to Non-`const` Object

One possible correction is to assign the constant string to a non-`const` object and use the non-`const` object as destination argument of `strchr`.

```
#include <string.h>
```

```
char buffer[] = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

Check Information

Group: Rule 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP40-C

Introduced in R2019a

CERT C: Rule EXP42-C

Do not compare padding data

Description

Rule Definition

Do not compare padding data.

Polyspace Implementation

This checker checks for **Memory comparison of padding data**.

Examples

Memory comparison of padding data

Issue

Memory comparison of padding data occurs when you use the `memcmp` function to compare two structures as a whole. In the process, you compare meaningless data stored in the structure padding.

For instance:

```
struct structType {
    char member1;
    int member2;
    .
    .
};

structType var1;
structType var2;
.
.
if(memcmp(&var1,&var2,sizeof(var1)))
{...}
```

Risk

If members of a structure have different data types, your compiler introduces additional padding for data alignment in memory. For an example of padding, see `Higher Estimate of Local Variable Size`.

The content of these extra padding bytes is meaningless. The C Standard allows the content of these bytes to be indeterminate, giving different compilers latitude to implement their own padding. If you perform a byte-by-byte comparison of structures with `memcmp`, you compare even the meaningless data stored in the padding. You might reach the false conclusion that two data structures are not equal, even if their corresponding members have the same value.

Fix

Instead of comparing two structures in one attempt, compare the structures member by member.

For efficient code, write a function that does the comparison member by member. Use this function for comparing two structures.

You can use `memcmp` for byte-by-byte comparison of structures only if you know that the structures do not contain padding. Typically, to prevent padding, you use specific attributes or pragmas such as `#pragma pack`. However, these attributes or pragmas are not supported by all compilers and make your code implementation-dependent. If your structures contain bit-fields, using these attributes or pragmas cannot prevent padding.

Example - Structures Compared with `memcmp`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{

    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    if (0 == memcmp(left, right, sizeof(S_Padding)))
    {
        return 1;
    }
    else
        return 0;
}
```

In this example, `memcmp` compares byte-by-byte the two structures that `left` and `right` point to. Even if the values stored in the structure members are the same, the comparison can show an inequality if the meaningless values in the padding bytes are not the same.

Correction — Compare Structures Member by Member

One possible correction is to compare individual structure members.

Note You can compare entire arrays by using `memcmp`. All members of an array have the same data type. Padding bytes are not required to store arrays.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{
    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    return ((left->c == right->c) &&
            (left->i == right->i) &&
            (left->bf1 == right->bf1) &&
            (left->bf2 == right->bf2) &&
            (memcmp(left->buffer, right->buffer, 20) == 0));
}
```

Check Information

Group: Rule 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP42-C

Introduced in R2019a

CERT C: Rule EXP43-C

Avoid undefined behavior when using restrict-qualified pointers

Description

Rule Definition

Avoid undefined behavior when using restrict-qualified pointers.

Polyspace Implementation

This checker checks for **Copy of overlapping memory**.

Examples

Copy of overlapping memory

Issue

Copy of overlapping memory occurs when there is a memory overlap between the source and destination argument of a copy function such as `memcpy` or `strcpy`. For instance, the source and destination arguments of `strcpy` are pointers to different elements in the same string.

Risk

If there is memory overlap between the source and destination arguments of copy functions, according to C standards, the behavior is undefined.

Fix

Determine if the memory overlap is what you want. If so, find an alternative function. For instance:

- If you are using `memcpy` to copy values from one memory location to another, use `memmove` instead of `memcpy`.
- If you are using `strcpy` to copy one string to another, use `memmove` instead of `strcpy`, as follows:

  ```
  s = strlen(source);
  memmove(destination, source, s + 1);
  ```

  `strlen` determines the string length without the null terminator. Therefore, you must move `s+1` bytes instead of `s` bytes.

Example - Overlapping Copy

```
#include <string.h>

char str[] = {"ABCDEFGH"};

void my_copy() {
    strcpy(&str[0],(const char*)&str[2]);
}
```

In this example, because the source and destination argument are pointers to the same string `str`, there is memory overlap between their allowed buffers.

Check Information

Group: Rule 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP43-C

Introduced in R2019a

CERT C: Rule EXP44-C

Do not rely on side effects in operands to sizeof, _Alignof, or _Generic

Description

Rule Definition

Do not rely on side effects in operands to sizeof, _Alignof, or _Generic.

Polyspace Implementation

This checker checks for **Side effect of expression ignored**.

Examples

Side effect of expression ignored

Issue

Side effect of expression ignored occurs when the `sizeof`, `_Alignof`, or `_Generic` operator operates on an expression with a side effect. When evaluated, an expression with side effect modifies at least one of the variables in the expression.

For instance, the defect checker does not flag `sizeof(n+1)` because `n+1` does not modify `n`. The checker flags `sizeof(n++)` because `n++` is intended to modify `n`.

The check also applies to the C++ operator `alignof` and its C extensions, `__alignof__` and `__typeof__`.

Risk

The expression in a `_Alignof` or `_Generic` operator is not evaluated. The expression in a `sizeof` operator is evaluated only if it is required for calculating the size of a variable-length array, for instance, `sizeof(a[n++])`.

When an expression with a side effect is not evaluated, the variable modification from the side effect does not happen. If you rely on the modification, you can see unexpected results.

Fix

Evaluate the expression with a side effect in a separate statement, and then use the result in a `sizeof`, `_Alignof`, or `_Generic` operator.

For instance, instead of:

```
a = sizeof(n++);
```

perform the operation in two steps:

```
n++;
a = sizeof(n);
```

The checker considers a function call as an expression with a side effect. Even if the function does not have side effects now, it might have side effects on later additions. The code is more maintainable if you call the function outside the `sizeof` operator.

Example - Increment Operator in `sizeof`

```
#include <stdio.h>

void func(void) {
    unsigned int a = 1U;
    unsigned int b = (unsigned int)sizeof(++a);
    printf ("%u, %u\n", a, b);
}
```

In this example, `sizeof` operates on `++a`, which is intended to modify `a`. Because the expression is not evaluated, the modification does not happen. The `printf` statement shows that `a` still has the value `1`.

Correction — Perform Increment Outside `sizeof`

One possible correction is to perform the increment first, and then provide the result to the `sizeof` operator.

```
#include <stdio.h>

void func(void) {
    unsigned int a = 1U;
    ++a;
    unsigned int b = (unsigned int)sizeof (a);
    printf ("%u, %u\n", a, b);
}
```

Check Information

Group: Rule 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP44-C

Introduced in R2019a

CERT C: Rule EXP45-C

Do not perform assignments in selection statements

Description

Rule Definition

Do not perform assignments in selection statements.

Polyspace Implementation

This checker checks for **Invalid use of = (assignment) operator**.

Examples

Invalid use of = (assignment) operator

Issue

Invalid use of = operator occurs when an assignment is made inside the predicate of a conditional, such as `if` or `while`.

In C and C++, a single equal sign is an assignment not a comparison. Using a single equal sign in a conditional statement can indicate a typo or a mistake.

Risk

- Conditional statement tests the wrong values— The single equal sign operation assigns the value of the right operand to the left operand. Then, because this assignment is inside the predicate of a conditional, the program checks whether the new value of the left operand is nonzero or not NULL.
- Maintenance and readability issues — Even if the assignment is intended, someone reading or updating the code can misinterpret the assignment as an equality comparison instead of an assignment.

Fix

- If the assignment is a bug, to check for equality, add a second equal sign (==).
- If the assignment inside the conditional statement was intentional, to improve readability, separate the assignment and the test. Move the assignment outside the control statement. In the control statement, simply test the result of the assignment.

  If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Single Equal Sign Inside an `if` Condition

```
#include <stdio.h>

void bad_equals_ex(int alpha, int beta)
{
    if(alpha = beta)
```

```
    {
        printf("Equal\n");
    }
}
```

The equal sign is flagged as a defect because the assignment operator is used within the predicate of the if-statement. The predicate assigns the value `beta` to `alpha`, then implicitly tests whether `alpha` is true or false.

Correction — Change Expression to Comparison

One possible correction is adding an additional equal sign. This correction changes the assignment to a comparison. The if condition compares whether `alpha` and `beta` are equal.

```
#include <stdio.h>

void equality_test(int alpha, int beta)
{
    if(alpha == beta)
    {
        printf("Equal\n");
    }
}
```

Correction — Assignment and Comparison Inside the `if` Condition

If an assignment must be made inside the predicate, a possible correction is adding an explicit comparison. This correction assigns the value of `beta` to `alpha`, then explicitly checks whether `alpha` is nonzero. The code is clearer.

```
#include <stdio.h>

int assignment_not_zero(int alpha, int beta)
{
    if((alpha = beta) != 0)
    {
        return alpha;
    }
    else
    {
        return 0;
    }
}
```

Correction — Move Assignment Outside the `if` Statement

If the assignment can be made outside the control statement, one possible correction is to separate the assignment and comparison. This correction assigns the value of `beta` to `alpha` before the if. Inside the if-condition, only `alpha` is given to test if `alpha` is nonzero or not NULL.

```
#include <stdio.h>

void assign_and_print(int alpha, int beta)
{
    alpha = beta;
    if(alpha)
    {
        printf("%d", alpha);
```

```
    }
}
```

Check Information

Group: Rule 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP45-C

Introduced in R2019a

CERT C: Rule EXP46-C

Do not use a bitwise operator with a Boolean-like operand

Description

Rule Definition

Do not use a bitwise operator with a Boolean-like operand.

Polyspace Implementation

This checker checks for **Possible invalid operation on boolean operand**.

Examples

Possible invalid operation on boolean operand

Issue

Possible invalid operation on boolean operand occurs when you use a Boolean operand in an arithmetic, relational, or bitwise operation and:

- The Boolean operand has a trap representation. The size of a Boolean type in memory is at least one addressable unit (size of `char`). A Boolean type requires only one bit to represent the value `true (1)` or `false (0)`. The representation of a Boolean operand in memory contains padding bits. The memory representation can result in values that are not `true` or `false`, a trap representation.
- The result of the operation can exceed the precision of the Boolean operand.

For example, in this code snippet:

```
bool_v >> 2
```

- If the value of `bool_v` is `true (1)` or `false (0)`, the bitwise shift exceeds the one-bit precision of `bool_v` and always results in `0`.
- If `bool_v` has a trap representation, the result of the operation is an arbitrary value.

Possible invalid operation on boolean operand raises no defect when:

- The operation does not result in a precision overflow. For instance, bitwise & or | operations with `0x01` or `0x00`.
- The Boolean operand cannot have a trap representation. For instance, a constant expression that results in `0` or `1`, or a comparison evaluated to `true` or `false`.

Risk

Arithmetic, relational, or bitwise operations on a Boolean operand can exceed the operand precision and cause unexpected results when used as a Boolean value. Operations on Boolean operands with trap representations can return arbitrary values.

Fix

Avoid performing operations on Boolean operands other than these operations:

- Assignment operation (=).
- Equality operations (== or !=).
- Logical operations (&&, ||, or !).

Example - Possible Trap Representation of Boolean Operand

```
#include <stdio.h>
#include <stdbool.h>

#define BOOL _Bool

int arr[2] = {1, 2};

int func(BOOL b)
{
    return arr[b];
}

int main(void)
{
    BOOL b;
    char* ptr = (char*)&b;
    *ptr = 64;
    return func(b);
}
```

In this example, Boolean operand `b` is used as an array index in `func` for an array with two elements. Depending on the compiler and optimization flags you use, the value `b` might not be `0` or `1`. For instance, in Linux Debian 8, if you use `gcc` version 4.9 with optimization flag `-O0`, the value of `b` is 64, which causes a buffer overflow.

Correction — Use Only Last Significant Bit Value of Boolean Operand

One possible correction is to use a variable `b0` of type `unsigned int` to get only the value of the last significant bit of the Boolean operand. The value of this bit is in the range `[0..1]`, even if the Boolean operand has a trap representation.

```
#include <stdio.h>
#include <stdbool.h>

#define BOOL _Bool

int arr[2] = {1, 2};

int func(BOOL b)
{
    unsigned int b0 = (unsigned int)b;
    b0 &= 0x1;
    return arr[b0];
}

int main(void)
{
```

```
    BOOL b;
    char* ptr = (char*)&b;
    *ptr = 64;
    return func(b);
}
```

Check Information

Group: Rule 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP46-C

Introduced in R2019a

CERT C: Rule EXP47-C

Do not call va_arg with an argument of the incorrect type

Description

Rule Definition

Do not call va_arg with an argument of the incorrect type.

Polyspace Implementation

This checker checks for these issues:

- **Incorrect data type passed to va_arg**.
- **Too many va_arg calls for current argument list**.

Examples

Incorrect data type passed to va_arg

Issue

Incorrect data type passed to va_arg when the data type in a `va_arg` call does not match the data type of the variadic function argument that `va_arg` reads.

For instance, you pass an `unsigned char` argument to a variadic function `func`. Because of default argument promotion, the argument is promoted to `int`. When you use a `va_arg` call that reads an `unsigned char` argument, a type mismatch occurs.

```
void func (int n, ...) {
   ...
   va_list args;
   va_arg(args, unsigned char);
   ...
}

void main(void) {
   unsigned char c;
   func(1,c);
}
```

Risk

In a variadic function (function with variable number of arguments), you use `va_arg` to read each argument from the variable argument list (`va_list`). The `va_arg` use does not guarantee that there actually exists an argument to read or that the argument data type matches the data type in the `va_arg` call. You have to make sure that both conditions are true.

Reading an incorrect type with a `va_arg` call can result in undefined behavior. Because function arguments reside on the stack, you might access an unwanted area of the stack.

Fix

Make sure that the data type of the argument passed to the variadic function matches the data type in the `va_arg` call.

Arguments of a variadic function undergo default argument promotions. The argument data types of a variadic function cannot be determined from a prototype. The arguments of such functions undergo default argument promotions (see Sec. 6.5.2.2 and 7.15.1.1 in the C99 Standard). Integer arguments undergo integer promotion and arguments of type `float` are promoted to `double`. For integer arguments, if a data type can be represented by an `int`, for instance, `char` or `short`, it is promoted to an `int`. Otherwise, it is promoted to an `unsigned int`. All other arguments do not undergo promotion.

To avoid undefined and implementation-defined behavior, minimize the use of variadic functions. Use the checkers for `MISRA C:2012 Rule 17.1` or `MISRA C++:2008 Rule 8-4-1` to detect use of variadic functions.

Example - `char` Used as Function Argument Type and `va_arg` argument

```
#include <stdarg.h>
#include <stdio.h>

unsigned char func(size_t count, ...) {
    va_list ap;
    unsigned char result = 0;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, unsigned char);
    }
    va_end(ap);
    return result;
}

void func_caller(void) {
    unsigned char c = 0x12;
    (void)func(1, c);
}
```

In this example, `func` takes an `unsigned char` argument, which undergoes default argument promotion to `int`. The data type in the `va_arg` call is still `unsigned char`, which does not match the `int` argument type.

Correction — Use `int` as `va_arg` Argument

One possible correction is to read an `int` argument with `va_arg`.

```
#include <stdarg.h>
#include <stdio.h>

unsigned char func(size_t count, ...) {
    va_list ap;
    unsigned char result = 0;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
    }
    va_end(ap);
    return result;
```

```
}

void func_caller(void) {
    unsigned char c = 0x12;
    (void)func(1, c);
}
```

Too many va_arg calls for current argument list

Issue

Too many va_arg calls for current argument list occurs when the number of calls to `va_arg` exceeds the number of arguments passed to the corresponding variadic function. The analysis raises a defect only when the variadic function is called.

Too many va_arg calls for current argument list does not raise a defect when:

- The number of calls to `va_arg` inside the variadic function is indeterminate. For example, if the calls are from an external source.
- The `va_list` used in `va_arg` is invalid.

Risk

When you call `va_arg` and there is no next argument available in `va_list`, the behavior is undefined. The call to `va_arg` might corrupt data or return an unexpected result.

Fix

Ensure that you pass the correct number of arguments to the variadic function.

Example - No Argument Available When Calling va_arg

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/
int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
        if (count > 0) {
/* No further argument available
* in va_list when calling va_arg
*/

            result += va_arg(ap, int);
        }
    }
    va_end(ap);
    return result;
}
```

```
void func(void) {

    (void)variadic_func(2, 100);

}
```

In this example, the named argument and only one variadic argument are passed to `variadic_func()` when it is called inside `func()`. On the second call to `va_arg`, no further variadic argument is available in `ap` and the behavior is undefined.

Correction — Pass Correct Number of Arguments to Variadic Function

One possible correction is to ensure that you pass the correct number of arguments to the variadic function.

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/

int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
        if (count > 0) {

/* The correct number of arguments is
* passed to va_list when variadic_func()
* is called inside func()
*/
            result += va_arg(ap, int);
        }
    }
    va_end(ap);
    return result;
}

void func(void) {

    (void)variadic_func(2, 100, 200);

}
```

Check Information

Group: Rule 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites
EXP47-C

Introduced in R2019a

CERT C: Rule INT30-C

Ensure that unsigned integer operations do not wrap

Description

Rule Definition

Ensure that unsigned integer operations do not wrap.

Polyspace Implementation

This checker checks for these issues:

- **Unsigned integer overflow**.
- **Unsigned integer constant overflow**.

Examples

Unsigned integer overflow

Issue

Unsigned integer overflow occurs when an operation on unsigned integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

The C11 standard states that unsigned integer overflows result in wrap-around behavior. However, a wrap around behavior might not always be desirable. For instance, if the result of a computation is used as an array size and the computation overflows, the array size is much smaller than expected.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling. In the error handling code, you can override the default wrap-around behavior for overflows and implement saturation behavior, for instance.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Add One to Maximum Unsigned Integer

```
#include <limits.h>

unsigned int plusplus(void) {

    unsigned uvar = UINT_MAX;
    uvar++;
    return uvar;
}
```

In the third statement of this function, the variable `uvar` is increased by 1. However, the value of `uvar` is the maximum unsigned integer value, so 1 plus the maximum integer value cannot be represented by an `unsigned int`. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `uvar` is reduced by modulo `UINT_MAX`. The result is `uvar = 1`.

Correction — Different Storage Type

One possible correction is to store the operation result in a larger data type. In this example, by returning an `unsigned long long` instead of an `unsigned int`, the overflow error is fixed.

```
#include <limits.h>

unsigned long long plusplus(void) {

    unsigned long long ullvar = UINT_MAX;
    ullvar++;
    return ullvar;
}
```

Unsigned integer constant overflow

Issue

Unsigned integer constant overflow occurs when you assign a compile-time constant to a unsigned integer variable whose data type cannot accommodate the value. An `n`-bit unsigned integer holds values in the range $[0, 2^n-1]$.

For instance, `c` is an 8-bit unsigned `char` variable that cannot hold the value 256.

```
unsigned char c = 256;
```

To determine the sizes of fundamental types, Bug Finder uses your specification for `Target processor type (-target)`.

Risk

The C standard states that overflowing unsigned integers must be wrapped around (see, for instance, the C11 standard, section 6.2.5). However, the wrap-around behavior can be unintended and cause unexpected results.

Fix

Check if the constant value is what you intended. If the value is correct, use a wider data type for the variable.

Example - Overflowing Constant from Macro Expansion

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_UNSIGNED_SHORT 65535

void main() {
    unsigned char c1 = MAX_UNSIGNED_CHAR + 1;
    unsigned short c2 = MAX_UNSIGNED_SHORT + 1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow.

Correction — Use Wider Data Type

One possible correction is to use a wider data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_UNSIGNED_SHORT 65535

void main() {
    unsigned short c1 = MAX_UNSIGNED_CHAR + 1;
    unsigned int c2 = MAX_UNSIGNED_SHORT + 1;
}
```

Check Information

Group: Rule 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT30-C

Introduced in R2019a

CERT C: Rule INT31-C

Ensure that integer conversions do not result in lost or misinterpreted data

Description

Rule Definition

Ensure that integer conversions do not result in lost or misinterpreted data.

Polyspace Implementation

This checker checks for these issues:

- **Integer conversion overflow**.
- **Call to memset with unintended value**.
- **Sign change integer conversion overflow**.
- **Tainted sign change conversion**.
- **Unsigned integer conversion overflow**.

Examples

Integer conversion overflow

Issue

Integer conversion overflow occurs when converting an integer to a smaller integer type. If the variable does not have enough bytes to represent the original value, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Integer conversion overflows result in undefined behavior.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Converting from `int` to `char`

```
char convert(void) {

    int num = 1000000;

    return (char)num;
}
```

In the return statement, the integer variable `num` is converted to a `char`. However, an 8-bit or 16-bit character cannot represent 1000000 because it requires at least 20 bits. So the conversion operation overflows.

Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number.

```
long convert(void) {

    int num = 1000000;

    return (long)num;
}
```

Call to memset with unintended value

Issue

Call to memset with unintended value occurs when Polyspace Bug Finder detects a use of the `memset` or `wmemset` function with possibly incorrect arguments.

`void *memset (void *ptr, int value, size_t num)` fills the first `num` bytes of the memory block that `ptr` points to with the specified `value`. If the argument `value` is incorrect, the memory block is initialized with an unintended value.

The unintended initialization can occur in the following cases.

Issue	Risk	Possible Fix
The second argument is `'0'` instead of `0` or `'\0'`.	The ASCII value of character `'0'` is 48 (decimal), `0x30` (hexadecimal), `069` (octal) but not `0` (or `'\0'`) .	If you want to initialize with `'0'`, use one of the ASCII values. Otherwise, use `0` or `'\0'`.
The second and third arguments are probably reversed. For instance, the third argument is a literal and the second argument is not a literal.	If the order is reversed, a memory block of unintended size is initialized with incorrect arguments.	Reverse the order of the arguments.

Issue	Risk	Possible Fix
The second argument cannot be represented in a byte.	If the second argument cannot be represented in a byte, and you expect each byte of a memory block to be filled with that argument, the initialization does not occur as intended.	Apply a bit mask to the argument to produce a wrapped or truncated result that can be represented in a byte. When you apply a bit mask, make sure that it produces an expected result. For instance, replace `memset(a, -13, sizeof(a))` with `memset(a, (-13) & 0xFF, sizeof(a))`.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Value Cannot Be Represented in a Byte

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE];
    int c = -2;
    memset(buf, (char)c, sizeof(buf));
}
```

In this example, `(char)c` cannot be represented in a byte.

Correction — Apply Cast

One possible correction is to apply a cast so that the result can be represented in a byte. However, check that the result of the cast is an acceptable initialization value.

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE   ];
    int c = -2;
    memset(buf, (unsigned char)c, sizeof(buf));
}
```

Sign change integer conversion overflow

Issue

Sign change integer conversion overflow occurs when converting an unsigned integer to a signed integer. If the variable does not have enough bytes to represent both the original constant and the sign bit, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Convert from `unsigned char` to `char`

```
char sign_change(void) {
    unsigned char count = 255;

    return (char)count;
}
```

In the return statement, the unsigned character variable `count` is converted to a signed character. However, `char` has 8 bits, 1 for the sign of the constant and 7 to represent the number. The conversion operation overflows because 255 uses 8 bits.

Correction — Change conversion types

One possible correction is using a larger integer type. By using an `int`, there are enough bits to represent the sign and the number value.

```
int sign_change(void) {
    unsigned char count = 255;

    return (int)count;
}
```

Tainted sign change conversion

Issue

Tainted sign change conversion looks for values from unsecure sources that are converted, implicitly or explicitly, from signed to unsigned values.

For example, functions that use `size_t` as arguments implicitly convert the argument to an unsigned integer. Some functions that implicitly convert `size_t` are:

```
bcmp
memcpy
```

```
memmove
strncmp
strncpy
calloc
malloc
memalign
```

Risk

If you convert a small negative number to unsigned, the result is a large positive number. The large positive number can create security vulnerabilities. For example, if you use the unsigned value in:

- Memory size routines — causes allocating memory issues.
- String manipulation routines — causes buffer overflow.
- Loop boundaries — causes infinite loops.

Fix

To avoid converting unsigned negative values, check that the value being converted is within an acceptable range. For example, if the value represents a size, validate that the value is not negative and less than the maximum value size.

Example - Set Memory Value with Size Argument

```
#include <stdlib.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedsignchange(int size) {
    char str[SIZE128] = "";
    if (size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

In this example, a `char` buffer is created and filled using `memset`. The size argument to `memset` is an input argument to the function.

The call to `memset` implicitly converts `size` to unsigned integer. If `size` is a large negative number, the absolute value could be too large to represent as an integer, causing a buffer overflow.

Correction — Check Value of `size`

One possible correction is to check if `size` is inside the valid range. This correction checks if `size` is greater than zero and less than the buffer size before calling `memset`.

```
#include <stdlib.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
```

```
};

void corrected_taintedsignchange(int size) {
    char str[SIZE128] = "";
    if (size>0 && size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

Unsigned integer conversion overflow

Issue

Unsigned integer conversion overflow occurs when converting an unsigned integer to a smaller unsigned integer type. If the variable does not have enough bytes to represent the original constant, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Integer conversion overflows result in undefined behavior.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Converting from `int` to `char`

```
unsigned char convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned char)unum;
}
```

In the return statement, the unsigned integer variable `unum` is converted to an unsigned character type. However, the conversion overflows because 1000000 requires at least 20 bits. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `unum` is reduced by modulo `2^8` because a character data type can only represent `2^8-1`.

Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number. For example, `long`.

```
unsigned long convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned long)unum;
}
```

Check Information

Group: Rule 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT31-C

Introduced in R2019a

CERT C: Rule INT32-C

Ensure that operations on signed integers do not result in overflow

Description

Rule Definition

Ensure that operations on signed integers do not result in overflow.

Polyspace Implementation

This checker checks for these issues:

- **Integer overflow**.
- **Tainted division operand**.
- **Tainted modulo operand**.

Examples

Integer overflow

Issue

Integer overflow occurs when an operation on integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`. W

Risk

Integer overflows on signed integers result in undefined behavior.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

To avoid overflows in general, try one of these techniques:

- Keep integer variable values restricted to within half the range of signed integers.

- In operations that might overflow, check for conditions that can lead to the overflow and implement wrap around or saturation behavior depending on how the result of the operation is used. The result then becomes predictable and can be safely used in subsequent computations.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Addition of Maximum Integer

```
#include <limits.h>

int plusplus(void) {

    int var = INT_MAX;
    var++;
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

Correction — Different Storage Type

One possible correction is to change data types. Store the result of the operation in a larger data type (Note that on a 32-bit machine, `int` and `long` has the same size). In this example, on a 32-bit machine, by returning a `long long` instead of an `int`, the overflow error is fixed.

```
#include <limits.h>

long long plusplus(void) {

    long long lvar = INT_MAX;
    lvar++;
    return lvar;
}
```

Tainted division operand

Issue

Tainted division operand detects division operations where one or both of the integer operands is from an unsecure source.

Risk

- If the numerator is the minimum possible value and the denominator is `-1`, your division operation overflows because the result cannot be represented by the current variable size.
- If the denominator is zero, your division operation fails possibly causing your program to crash.

These risks can be used to execute arbitrary code. This code is usually outside the scope of a program's implicit security policy.

Fix

Before performing the division, validate the values of the operands. Check for denominators of `0` or `-1`, and numerators of the minimum integer value.

Example - Division of Function Arguments

```
extern void print_int(int);

int taintedintdivision(int usernum, int userden) {
    int r =  usernum/userden;
    print_int(r);
    return r;
}
```

This example function divides two argument variables, then prints and returns the result. The argument values are unknown and can cause division by zero or integer overflow.

Correction — Check Values

One possible correction is to check the values of the numerator and denominator before performing the division.

```
#include "limits.h"

extern void print_int(int);

int taintedintdivision(int usernum, int userden) {
    int r = 0;
    if (userden!=0 && !(usernum=INT_MIN && userden==-1)) {
        r = usernum/userden;
    }
    print_int(r);
    return r;
}
```

Tainted modulo operand

Issue

Tainted modulo operand checks the operands of remainder `%` operations. Bug Finder flags modulo operations with one or more tainted operands.

Risk

- If the second remainder operand is zero, your remainder operation fails, causing your program to crash.
- If the second remainder operand is `-1`, your remainder operation can overflow if the remainder operation is implemented based on the division operation that can overflow.
- If one of the operands is negative, the operation result is uncertain. For C89, the modulo operation is not standardized, so the result from negative operands is implementation-defined.

These risks can be exploited by attackers to gain access to your program or the target in general.

Fix

Before performing the modulo operation, validate the values of the operands. Check the second operand for values of `0` and `-1`. Check both operands for negative values.

Example - Modulo with Function Arguments

```
extern void print_int(int);
```

```
int taintedintmod(int userden) {
    int rem =  128%userden;
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using an input argument. The argument is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

Correction — Check Operand Values

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
extern void print_int(int);

int taintedintmod(int userden) {
    int rem = 0;
    if (userden > 0) {
        rem = 128 % userden;
    }
    print_int(rem);
    return rem;
}
```

Check Information

Group: Rule 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT32-C

Introduced in R2019a

CERT C: Rule INT33-C

Ensure that division and remainder operations do not result in divide-by-zero errors

Description

Rule Definition

Ensure that division and remainder operations do not result in divide-by-zero errors.

Polyspace Implementation

This checker checks for these issues:

- **Integer division by zero**.
- **Tainted division operand**.
- **Tainted modulo operand**.

Examples

Integer division by zero

Issue

Integer division by zero occurs when the denominator of a division or modulo operation can be a zero-valued integer.

Risk

A division by zero can result in a program crash.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the denominator variable acquires a zero value. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

It is a good practice to check for zero values of a denominator before division and handle the error. Instead of performing the division directly:

```
res = num/den;
```

use a library function that handles zero values of the denominator before performing the division:

```
res = div(num, den);
```

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Dividing an Integer by Zero

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    result = num/denom;

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

Correction — Check Before Division

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    if (denom != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
int fraction(int num)
{
    int denom = 2;
    int result = 0;

    result = num/denom;

    return result;
}
```

Example - Modulo Operation with Zero

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        arr[i] = input % i;
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

In this example, Polyspace flags the modulo operation as a division by zero. Because modulo is inherently a division operation, the divisor (right hand argument) cannot be zero. The modulo

operation uses the `for` loop index as the divisor. However, the `for` loop starts at zero, which cannot be an iterator.

Correction — Check Divisor Before Operation

One possible correction is checking the divisor before the modulo operation. In this example, see if the index `i` is zero before the modulo operation.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        if(i != 0)
        {
             arr[i] = input % i;
        }
        else
        {
             arr[i] = input;
        }
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

Correction — Change Divisor

Another possible correction is changing the divisor to a nonzero integer. In this example, add one to the index before the `%` operation to avoid dividing by zero.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
         arr[i] = input % (i+1);
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

Tainted division operand

Issue

Tainted division operand detects division operations where one or both of the integer operands is from an unsecure source.

Risk

- If the numerator is the minimum possible value and the denominator is `-1`, your division operation overflows because the result cannot be represented by the current variable size.
- If the denominator is zero, your division operation fails possibly causing your program to crash.

These risks can be used to execute arbitrary code. This code is usually outside the scope of a program's implicit security policy.

Fix

Before performing the division, validate the values of the operands. Check for denominators of `0` or `-1`, and numerators of the minimum integer value.

Example - Division of Function Arguments

```
extern void print_int(int);

int taintedintdivision(int usernum, int userden) {
    int r =  usernum/userden;
    print_int(r);
    return r;
}
```

This example function divides two argument variables, then prints and returns the result. The argument values are unknown and can cause division by zero or integer overflow.

Correction — Check Values

One possible correction is to check the values of the numerator and denominator before performing the division.

```
#include "limits.h"

extern void print_int(int);

int taintedintdivision(int usernum, int userden) {
    int r = 0;
    if (userden!=0 && !(usernum=INT_MIN && userden==-1)) {
        r = usernum/userden;
    }
    print_int(r);
    return r;
}
```

Tainted modulo operand

Issue

Tainted modulo operand checks the operands of remainder `%` operations. Bug Finder flags modulo operations with one or more tainted operands.

Risk

- If the second remainder operand is zero, your remainder operation fails, causing your program to crash.
- If the second remainder operand is `-1`, your remainder operation can overflow if the remainder operation is implemented based on the division operation that can overflow.
- If one of the operands is negative, the operation result is uncertain. For C89, the modulo operation is not standardized, so the result from negative operands is implementation-defined.

These risks can be exploited by attackers to gain access to your program or the target in general.

Fix

Before performing the modulo operation, validate the values of the operands. Check the second operand for values of `0` and `-1`. Check both operands for negative values.

Example - Modulo with Function Arguments

```
extern void print_int(int);

int taintedintmod(int userden) {
    int rem =  128%userden;
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using an input argument. The argument is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

Correction — Check Operand Values

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
extern void print_int(int);

int taintedintmod(int userden) {
    int rem = 0;
    if (userden > 0) {
        rem = 128 % userden;
    }
    print_int(rem);
    return rem;
}
```

Check Information

Group: Rule 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT33-C

Introduced in R2019a

CERT C: Rule INT34-C

Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand

Description

Rule Definition

Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand.

Polyspace Implementation

This checker checks for these issues:

- **Shift of a negative value**.
- **Shift operation overflow**.

Examples

Shift of a negative value

Issue

Shift of a negative value occurs when a bit-wise shift is used on a variable that can have negative values.

Risk

Shifts on negative values overwrite the sign bit that identifies a number as negative. The shift operation can result in unexpected values.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variable being shifted acquires negative values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

To fix the defect, check for negative values before the bit-wise shift operation and perform appropriate error handling.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Shifting a negative variable

```
int shifting(int val)
{
    int res = -1;
```

```
    return res << val;
}
```

In the return statement, the variable `res` is shifted a certain number of bits to the left. However, because `res` is negative, the shift might overwrite the sign bit.

Correction — Change the Data Type

One possible correction is to change the data type of the shifted variable to unsigned. This correction eliminates the sign bit, so left shifting does not change the sign of the variable.

```
int shifting(int val)
{
    unsigned int res = -1;
    return res << val;
}
```

Shift operation overflow

Issue

Shift operation overflow occurs when a shift operation can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Shift operation overflows can result in undefined behavior.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the shift operation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the shift operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Left Shift of Integer

```
int left_shift(void) {

    int foo = 33;
    return 1 << foo;
}
```

In the return statement of this function, bit-wise shift operation is performed shifting 1 `foo` bits to the left. However, an `int` has only 32 bits, so the range of the shift must be between 0 and 31. Therefore, this shift operation causes an overflow.

Correction — Different storage type

One possible correction is to store the shift operation result in a larger data type. In this example, by returning a `long long` instead of an `int`, the overflow defect is fixed.

```
long long left_shift(void) {

    int foo = 33;
    return 1LL << foo;
}
```

Check Information

Group: Rule 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT34-C

Introduced in R2019a

CERT C: Rule INT35-C

Use correct integer precisions

Description

Rule Definition

Use correct integer precisions.

Polyspace Implementation

This checker checks for **Integer precision exceeded**.

Examples

Integer precision exceeded

Issue

Integer precision exceeded occurs when an integer expression uses the integer size in an operation that exceeds the integer precision. On some architectures, the size of an integer in memory can include sign and padding bits. On these architectures, the integer size is larger than the precision which is just the number of bits that represent the value of the integer.

Risk

Using the size of an integer in an operation on the integer precision can result in integer overflow, wrap around, or unexpected results. For instance, an unsigned integer can be stored in memory in 64 bits, but uses only 48 bits to represent its value. A 56 bits left-shift operation on this integer is undefined behavior.

Assuming that the size of an integer is equal to its precision can also result in program portability issues between different architectures.

Fix

Do not use the size of an integer instead of its precision. To determine the integer precision, implement a precision computation routine or use a builtin function such as `__builtin_popcount()`.

Example - Using Size of `unsigned int` for Left Shift Operation

```
#include <limits.h>

unsigned int func(unsigned int exp)
{
    if (exp >= sizeof(unsigned int) * CHAR_BIT) {
        /* Handle error */
    }
    return 1U << exp;
}
```

In this example, the function uses a left shift operation to return the value of 2 raised to the power of `exp`. The operation shifts the bits of `1U` by `exp` positions to the left. The `if` statement ensures that

the operation does not shift the bits by a number of positions `exp` greater than the size of an `unsigned int`. However, if `unsigned int` contains padding bits, the value returned by `sizeof()` is larger than the precision of `unsigned int`. As a result, some values of `exp` might be too large, and the shift operation might be undefined behavior.

Correction — Implement Function to Compute Precision of `unsigned int`

One possible correction is to implement a function `popcount()` that computes the precision of `unsigned int` by counting the number of set bits.

```
#include <stddef.h>
#include <stdint.h>
#include <limits.h>

size_t popcount(uintmax_t);
#define PRECISION(umax_value) popcount(umax_value)


unsigned int func(unsigned int exp)
{
    if (exp >= PRECISION(UINT_MAX)) {
        /* Handle error */
    }
    return 1 << exp;
}




size_t popcount(uintmax_t num)
{
    size_t precision = 0;
    while (num != 0) {
        if (num % 2 == 1) {
            precision++;
        }
        num >>= 1;
    }
    return precision;
}
```

Check Information

Group: Rule 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT35-C

Introduced in R2019a

CERT C: Rule INT36-C

Converting a pointer to integer or integer to pointer

Description

Rule Definition

Converting a pointer to integer or integer to pointer.

Polyspace Implementation

This checker checks for **Unsafe conversion between pointer and integer**.

Examples

Unsafe conversion between pointer and integer

Issue

Unsafe conversion between pointer and integer checks for pointer to integer and integer to pointers conversions. If you convert between a pointer, `intptr_t`, or `uintprt_t` and an integer type, such as `enum`, `ptrdiff_t`, or `pid_t`, Polyspace raises a defect.

Risk

The mapping between pointers and integers is not always consistent with the addressing structure of the environment.

Converting from pointers to integers can create:

- Truncated or out of range integer values.
- Invalid integer types.

Converting from integers to pointers can create:

- Misaligned pointers or misaligned objects.
- Invalid pointer addresses.

Fix

Where possible, avoid pointer-to-integer or integer-to-pointer conversions. If you want to convert a `void` pointer to an integer, so that you do not change the value, use types:

- C99 — `intptr_t` or `uintptr_t`
- C90 — `size_t` or `ssize_t`

Example - Integer to Pointer Conversions

```
unsigned int *badintptrcast(void)
{
    unsigned int *ptr0 = (unsigned int *)0xdeadbeef;
    char *ptr1 = (char *)0xdeadbeef;
```

```
    return (unsigned int *)(ptr0 - (unsigned int *)ptr1);
}
```

In this example, there are three conversions, two unsafe conversions and one safe conversion. The first conversion of `0xdeadbeef` to `unsigned int*` causes alignment issues for the pointer. The second conversion of `0xdeadbeef` to `char *` is safe because there are no alignment issues for `char`. The third conversion in the return casts `ptrdiff_t` to a pointer. This pointer might or might not point to an invalid address.

Correction — Use `intptr_t`

One possible correction is to use `intptr_t` types to store the pointer address `0xdeadbeef`. Also, you can change the second pointer to an integer offset so that there is no longer a conversion from `ptrdiff_t` to a pointer.

```
#include <stdint.h>

unsigned int *badintptrcast(void)
{
    intptr_t iptr0 = (intptr_t)0xdeadbeef;
    int offset = 0;
    return (unsigned int *)(iptr0 - offset);
}
```

Check Information

Group: Rule 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT36-C

Introduced in R2019a

CERT C: Rule FLP30-C

Do not use floating-point variables as loop counters

Description

Rule Definition

Do not use floating-point variables as loop counters.

Polyspace Implementation

This checker checks for **Use of float variable as loop counter**.

Examples

Use of float variable as loop counter

Issue

The issue occurs when a loop counter has a floating type.

If the `for` index is a variable symbol, Polyspace checks that it is not a float.

Risk

When using a floating-point loop counter, accumulation of rounding errors can result in a mismatch between the expected and actual number of iterations. This rounding error can happen when a loop step that is not a power of the floating point radix is rounded to a value that can be represented by a float.

Even if a loop with a floating-point loop counter appears to behave correctly on one implementation, it can give a different number of iteration on another implementation.

Example - `for` Loop Counters

```
int main(void){
    unsigned int counter = 0u;
    int result = 0;
    float foo;

    // Float loop counters
    for(float foo = 0.0f; foo < 1.0f; foo +=0.001f){
        /* Non-compliant - counter = 1000 at the end of the loop */
        ++counter;
    }

    float fff = 0.0f;
    for(fff = 0.0f; fff <12.0f; fff += 1.0f){    /* Non-compliant*/
        result++;
    }

    // Integer loop count
    for(unsigned int count = 0u; count < 1000u; ++count){ /* Compliant */
```

```
        foo = (float) count * 0.001f;
    }
}
```

In this example, the three `for` loops show three different loop counters. The first and second `for` loops use float variables as loop counters, and therefore are not compliant. The third loop uses the integer `count` as the loop counter. Even though `count` is used as a float inside the loop, the variable remains an integer when acting as the loop index. Therefore, this `for` loop is compliant.

Example - `while` Loop Counters

```
int main(void){
    unsigned int u32a;
    float foo;

    foo = 0.0f;
    while (foo < 1.0f){
        foo += 0.001f;  /* Non-compliant - foo used as a loop counter */
    }

    foo = read_float32();
    do{
        u32a = read_u32();
    }while( ((float)u32a - foo) > 10.0f );
                        /* Compliant - foo doesn't change in the loop */
                        /*  so cannot be a counter */
    return 1;
}
```

This example shows two `while` loops both of which use `foo` in the `while`-loop conditions.

The first `while` loop uses `foo` in the condition and inside the loop. Because `foo` changes, floating-point rounding errors can cause unexpected behavior.

The second `while` loop does not use `foo` inside the loop, but does use `foo` inside the `while`-condition. So `foo` is not the loop counter. The integer `u32a` is the loop counter because it changes inside the loop and is part of the while condition. Because `u32a` is an integer, the rounding error issue is not a concern, making this `while` loop compliant.

Check Information

Group: Rule 05. Floating Point (FLP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FLP30-C

Introduced in R2019a

CERT C: Rule FLP32-C

Prevent or detect domain and range errors in math functions

Description

Rule Definition

Prevent or detect domain and range errors in math functions.

Polyspace Implementation

This checker checks for **Invalid use of standard library floating point routine**.

Examples

Invalid use of standard library floating point routine

Issue

Invalid use of standard library floating point routine occurs when you use invalid arguments with a floating point function from the standard library. This defect picks up:

- Rounding and absolute value routines

  ```
  ceil, fabs, floor, fmod
  ```
- Fractions and division routines

  ```
  fmod, modf
  ```
- Exponents and log routines

  ```
  frexp, ldexp, sqrt, pow, exp, log, log10
  ```
- Trigonometry function routines

  ```
  cos, sin, tan, acos, asin, atan, atan2, cosh, sinh, tanh, acosh, asinh,
  atanh
  ```

Risk

Domain errors on standard library floating point functions result in implementation-defined values. If you use the function return value in subsequent computations, you can see unexpected results.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the function argument acquires invalid values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

It is a good practice to handle for domain errors before using a standard library floating point function. For instance, before calling the `acos` function, check if the argument is in [-1.0, 1.0] and handle the error.

See examples of fixes below.

If you do not want to fix the issue, for instance, when you handle infinities in your code, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Arc Cosine Operation

```
#include <math.h>

double arccosine(void) {
    double degree = 5.0;
    return acos(degree);
}
```

The input value to `acos` must be in the interval `[-1,1]`. This input argument, `degree`, is outside this range.

Correction — Change Input Argument

One possible correction is to change the input value to fit the specified range. In this example, change the input value from degrees to radians to fix this defect.

```
#include <math.h>

double arccosine(void) {
    double degree = 5.0;
    double radian = degree * 3.14159 / 180.;
    return acos(radian);
}
```

Check Information

Group: Rule 05. Floating Point (FLP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FLP32-C

Introduced in R2019a

CERT C: Rule FLP34-C

Ensure that floating-point conversions are within range of the new type

Description

Rule Definition

Ensure that floating-point conversions are within range of the new type.

Polyspace Implementation

This checker checks for **Float conversion overflow**.

Examples

Float conversion overflow

Issue

Float conversion overflow occurs when converting a floating point number to a smaller floating point data type. If the variable does not have enough memory to represent the original number, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Overflows can result in unpredictable values from computations. The result can be infinity or the maximum finite value depending on the rounding mode used in the implementation. If you use the result of an overflowing conversion in subsequent computations and do not account for the overflow, you can see unexpected results.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variable being converted acquires its current value You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller floating point types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Converting from `double` to `float`

```
float convert(void) {

    double diam = 1e100;
    return (float)diam;
}
```

In the return statement, the variable `diam` of type double (64 bits) is converted to a variable of type float (32 bits). However, the value 1^100 requires more than 32 bits to be precisely represented.

Check Information

Group: Rule 05. Floating Point (FLP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FLP34-C

Introduced in R2019a

CERT C: Rule FLP36-C

Preserve precision when converting integral values to floating-point type

Description

Rule Definition

Preserve precision when converting integral values to floating-point type.

Polyspace Implementation

This checker checks for **Precision loss in integer to float conversion**.

Examples

Precision loss in integer to float conversion

Issue

Precision loss from integer to float conversion occurs when you cast an integer value to a floating-point type that cannot represent the original integer value.

For instance, the `long int` value `1234567890L` is too large for a variable of type `float` .

Risk

If the floating-point type cannot represent the integer value, the behavior is undefined (see C11 standard, 6.3.1.4, paragraph 2). For instance, least significant bits of the variable value can be dropped leading to unexpected results.

Fix

Convert to a floating-point type that can represent the integer value.

For instance, if the `float` data type cannot represent the integer value, use the `double` data type instead.

When writing a function that converts an integer to floating point type, before the conversion, check if the integer value can be represented in the floating-point type. For instance, `DBL_MANT_DIG * log2(FLT_RADIX)` represents the number of base-2 digits in the type `double`. Before conversion to the type `double`, check if this number is greater than or equal to the precision of the integer that you are converting. To determine the precision of an integer `num`, use this code:

```
size_t precision = 0;
while (num != 0) {
   if (num % 2 == 1) {
     precision++;
   }
   num >>= 1;
}
```

Some implementations provide a builtin function to determine the precision of an integer. For instance, GCC provides the function `__builtin_popcount`.

Example - Conversion of Large Integer to Floating-Point Type

```
#include <stdio.h>

int main(void) {
  long int big = 1234567890L;
  float approx = big;
  printf("%ld\n", (big - (long int)approx));
  return 0;
}
```

In this example, the `long int` variable `big` is converted to `float`.

Correction — Use a Wider Floating-Point Type

One possible correction is to convert to the `double` data type instead of `float`.

```
#include <stdio.h>

int main(void) {
  long int big = 1234567890L;
  double approx = big;
  printf("%ld\n", (big - (long int)approx));
  return 0;
}
```

Check Information

Group: Rule 05. Floating Point (FLP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FLP36-C

Introduced in R2019a

CERT C: Rule FLP37-C

Do not use object representations to compare floating-point values

Description

Rule Definition

Do not use object representations to compare floating-point values.

Polyspace Implementation

This checker checks for **Memory comparison of float-point values**.

Examples

Memory comparison of float-point values

Issue

Memory comparison of float-point values occurs when you compare the object representation of floating-point values or the object representation of structures containing floating-point members. When you use the functions `memcmp`, `bcmp`, or `wmemcmp` to perform the bit pattern comparison, the defect is raised.

Risk

The object representation of floating-point values uses specific bit patterns to encode those values. Floating-point values that are equal, for instance `-0.0` and `0.0` in the IEC 60559 standard, can have different bit patterns in their object representation. Similarly, floating-point values that are not equal can have the same bit pattern in their object representation.

Fix

When you compare structures containing floating-point members, compare the structure members individually.

To compare two floating-point values, use the `==` or `!=` operators. If you follow a standard that discourages the use of these operators, such as MISRA, ensure that the difference between the floating-point values is within an acceptable range.

Example - Using `memcmp` to Compare Structures with Floating-Point Members

```
#include <string.h>

typedef struct {
    int i;
    float f;
} myStruct;

extern void initialize_Struct(myStruct *);

int func_cmp(myStruct *s1, myStruct *s2) {
/* Comparison between structures containing
```

```
* floating-point members */
    return memcmp
        ((const void *)s1, (const void *)s2, sizeof(myStruct));
}

void func(void) {
    myStruct s1, s2;
    initialize_Struct(&s1);
    initialize_Struct(&s2);
    (void)func_cmp(&s1, &s2);
}
```

In this example, `func_cmp()` calls `memcmp()` to compare the object representations of structures `s1` and `s2`. The comparison might be inaccurate because the structures contain floating-point members.

Correction — Compare Structure Members Individually

One possible correction is to compare the structure members individually and to ensure that the difference between the floating-point values is within an acceptable range defined by `ESP`.

```
 #include <string.h>

typedef struct {
    int i;
    float f;
} myStruct;

extern void initialize_Struct(myStruct *);

#define ESP 0.00001

int func_cmp(myStruct *s1, myStruct *s2) {

/*Structure members are compared individually */
    return ((s1->i == s2->i) &&
            (fabsf(s1->f - s2->f) <= ESP));
}

void func(void) {
    myStruct s1, s2;
    initialize_Struct(&s1);
    initialize_Struct(&s2);
    (void)func_cmp(&s1, &s2);
}
```

Check Information

Group: Rule 05. Floating Point (FLP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FLP37-C

Introduced in R2019a

CERT C: Rule ARR30-C

Do not form or use out-of-bounds pointers or array subscripts

Description

Rule Definition

Do not form or use out-of-bounds pointers or array subscripts.

Polyspace Implementation

This checker checks for these issues:

- **Array access out of bounds**.
- **Pointer access out of bounds**.
- **Array access with tainted index**.
- **Use of tainted pointer**.
- **Pointer dereference with tainted offset**.

Examples

Array access out of bounds

Issue

Array access out of bounds occurs when an array index falls outside the range `[0...array_size-1]` during array access.

Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back

using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Array Access Out of Bounds Error

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

Correction — Keep Array Index Within Array Bounds

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

Pointer access out of bounds

Issue

Pointer access out of bounds occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

Array access with tainted index

Issue

Array access with tainted index detects reading or writing to an array by using a tainted index that has not been validated.

Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

Example - Use Index to Return Buffer Value

```
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    return tab[num];
}
```

In this example, the index `num` accesses the array `tab`. The function does not check to see if `num` is inside the range of `tab`.

Correction — Check Range Before Use

One possible correction is to check that `num` is in range before using it.

```
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    if (num >= 0 && num < SIZE100) {
           return tab[num];
    } else {
        return -9999;
    }
}
```

Use of tainted pointer

Issue

Use of tainted pointer defect is raised when:

- Tainted NULL pointer — the pointer is not validated against NULL.
- Tainted size pointer — the size of the memory zone that a pointer points to is not validated.

Note On a single pointer, your code can have instances of **Use of tainted pointer**, **Pointer dereference with tainted offset**, and **Tainted NULL or non-null-terminated string**. Bug Finder raises only the first tainted pointer defect that it finds.

Risk

An attacker can give your program a pointer that points to unexpected memory locations. If the pointer is dereferenced to write, the attacker can:

- Modify the state variables of a critical program.
- Cause your program to crash.
- Execute unwanted code.

If the pointer is dereferenced to read, the attacker can:

- Read sensitive data.
- Cause your program to crash.
- Modify a program variable to an unexpected value.

Fix

Avoid use of pointers from external sources.

Alternatively, if you trust the external source, sanitize the pointer before dereference. In a separate sanitization function:

- Check that the pointer is not NULL.
- Check the size of the memory location (if possible). This second check validates whether the size of the data the pointer points to matches the size your program expects.

The defect still appears in the body of the sanitization function. However, if you use a sanitization function, instead of several occurrences, the defect appears only once. You can justify the defect and

hide it in later reviews by using code annotations. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Function That Dereferences an External Pointer

```
void taintedptr(int* p, int i) {
    *p = i;
}
```

In this example, the pointer `*p` is passed as an argument, and the value is changed. The pointer can be null or point to unknown memory, which can be vulnerable.

Correction — Avoid Use of External Pointers

One possible correction is to avoid pointers from external sources.

```
int *taintedptr(int i) {
    /* Use heap memory allocated in the application */
    int *p = (int *)malloc(sizeof (int));
    if (p != NULL) { /* Check for success */
        *p = i;
    }
return p;
}
```

Correction — Check Pointer

Another possible correction is to sanitize the pointer before using it. This example uses a second function to check if the pointer is null and can be dereferenced.

```
#include <stdlib.h>

int* sanitize_ptr(int* p) {
    int* res = NULL;
    if (p && *p) { /* Tainted pointer detected here, used as "firewall" */
        /* Pointer is not null and dereference ok */
        res = p;
    }
    return res;
}
void taintedptr(int* p, int i) {
    p = sanitize_ptr(p);
    if (p) {
        *p = i;
    }
}
```

Pointer dereference with tainted offset

Issue

Pointer dereference with tainted offset detects pointer dereferencing, either reading or writing, using an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

Fix

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

Example - Dereference Pointer Array

```
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(int i) {
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[i];
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `i`. The value of `i` could be outside the pointer range, causing an out-of-range error.

Correction — Check Index Before Dereference

One possible correction is to validate the value of the index. If the index is inside the valid range, continue with the pointer dereferencing.

```
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);
```

```
int taintedptroffset(int i) {
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (i>0 && i<SIZE10) {
            c = pint[i];
        }
        free(pint);
    }
    return c;
}
```

Check Information

Group: Rule 06. Arrays (ARR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

ARR30-C

Introduced in R2019a

CERT C: Rule ARR32-C

Ensure size arguments for variable length arrays are in a valid range

Description

Rule Definition

Ensure size arguments for variable length arrays are in a valid range.

Polyspace Implementation

This checker checks for these issues:

- **Memory allocation with tainted size**.
- **Tainted size of variable length array**.

Examples

Memory allocation with tainted size

Issue

Memory allocation with tainted size checks memory allocation functions, such as `calloc` or `malloc`, for size arguments from unsecured sources.

Risk

Uncontrolled memory allocation can cause your program to request too much system memory. This consequence can lead to a crash due to an out-of-memory condition, or assigning too many resources.

Fix

Before allocating memory, check the value of your arguments to check that they do not exceed the bounds.

Example - Allocate Memory Using Input Argument

```
#include "stdlib.h"

int* bug_taintedmemoryallocsize(size_t size) {
    int* p = (int*)malloc(size);
    return p;
}
```

In this example, `malloc` allocates `size` amount of memory for the pointer `p`. `size` is an outside variable, so could be any size value. If the size is larger than the amount of memory you have available, your program could crash.

Correction — Check Size of Memory to be Allocated

One possible correction is to check the size of the memory that you want to allocate before performing the `malloc` operation. This example checks to see if the size is positive and less than the maximum size.

```
#include "stdlib.h"

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int* corrected_taintedmemoryallocsize(int size) {
    int* p = NULL;
    if (size>0 && size<SIZE128) {          /* Fix: Check entry range before use */
        p = (int*)malloc((unsigned int)size);
    }
    return p;
}
```

Tainted size of variable length array

Issue

Tainted size of variable length array detects variable length arrays (VLA) whose size is from an unsecure source.

Risk

If an attacker changed the size of your VLA to an unexpected value, it can cause your program to crash or behave unexpectedly.

If the size is non-positive, the behavior of the VLA is undefined. Your program does not perform as expected.

If the size is unbounded, the VLA can cause memory exhaustion or stack overflow.

Fix

Validate your VLA size to make sure that it is positive and less than a maximum value.

Example - Input Argument Used as Size of VLA

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedvlasize(int size) {

    int tabvla[size];
    int res = 0;
    for (int i=0 ; i<SIZE10 ; ++i) {
        tabvla[i] = i*i;
        res += tabvla[i];
    }
    return res;
}
```

In this example, a variable length array size is based on an input argument. Because this input argument value is not checked, the size may be negative or too large.

Correction — Check VLA Size

One possible correction is to check the size variable before creating the variable length array. This example checks if the size is larger than 10 and less than 100, before creating the VLA

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedvlasize(int size) {
    int res = 0;
    if (size>SIZE10 && size<SIZE100) {
        int tabvla[size];
        for (int i=0 ; i<SIZE10 ; ++i) {
            tabvla[i] = i*i;
            res += tabvla[i];
        }
    }
    return res;
}
```

Check Information

Group: Rule 06. Arrays (ARR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

ARR32-C

Introduced in R2019a

CERT C: Rule ARR36-C

Do not subtract or compare two pointers that do not refer to the same array

Description

Rule Definition

Do not subtract or compare two pointers that do not refer to the same array.

Polyspace Implementation

This checker checks for **Subtraction or comparison between pointers to different arrays**.

Examples

Subtraction or comparison between pointers to different arrays

Issue

Subtraction or comparison between pointers to different arrays occurs when you subtract or compare pointers that are null or that point to elements in different arrays. The relational operators for the comparison are >, <, >=, and <=.

Risk

When you subtract two pointers to elements in the same array, the result is the difference between the subscripts of the two array elements. Similarly, when you compare two pointers to array elements, the result is the positions of the pointers relative to each other. If the pointers are null or point to different arrays, a subtraction or comparison operation is undefined. If you use the subtraction result as a buffer index, it can cause a buffer overflow.

Fix

Before you subtract or use relational operators to compare pointers to array elements, check that they are non-null and that they point to the same array.

Example - Subtraction Between Pointers to Elements in Different Arrays

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE20 20

size_t func(void)
{
    int nums[SIZE20];
    int end;
    int *next_num_ptr = nums;
    size_t free_elements;
    /* Increment next_num_ptr as array fills */

    /* Subtraction operation is undefined unless array nums
    is adjacent to variable end in memory. */
```

```
    free_elements = &end - next_num_ptr;
    return free_elements;
}
```

In this example, the array `nums` is incrementally filled. Pointer subtraction is then used to determine how many free elements remain. Unless `end` points to a memory location one past the last element of `nums`, the subtraction operation is undefined.

Correction — Subtract Pointers to the Same Array

Subtract the pointer to the last element that was filled from the pointer to the last element in the array.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE20 20

size_t func(void)
{
    int nums[SIZE20];
    int *next_num_ptr = nums;
    size_t free_elements;
    /* Increment next_num_ptr as array fills */

    /* Subtraction operation involves pointers to the same array. */
    free_elements = &(nums[SIZE20 - 1]) - next_num_ptr;

    return free_elements + 1;
}
```

Check Information

Group: Rule 06. Arrays (ARR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

ARR36-C

Introduced in R2019a

CERT C: Rule ARR37-C

Do not add or subtract an integer to a pointer to a non-array object

Description

Rule Definition

Do not add or subtract an integer to a pointer to a non-array object.

Polyspace Implementation

This checker checks for **Invalid assumptions about memory organization**.

Examples

Invalid assumptions about memory organization

Issue

Invalid assumptions about memory organization occurs when you compute the address of a variable in the stack by adding or subtracting from the address of another non-array variable.

Risk

When you compute the address of a variable in the stack by adding or subtracting from the address of another variable, you assume a certain memory organization. If your assumption is incorrect, accessing the computed address can be invalid.

Fix

Do not perform an access that relies on assumptions about memory organization.

Example - Reliance on Memory Organization

```
void func(void) {
    int var1 = 0x00000011, var2;
    *(&var1 + 1) = 0;
}
```

In this example, the programmer relies on the assumption that `&var1 + 1` provides the address of `var2`. Therefore, an **Invalid assumptions about memory organization** appears on the `+` operation. In addition, a **Pointer access out of bounds** error also appears on the dereference.

Correction — Do Not Rely on Memory Organization

One possible correction is not perform direct computation on addresses to access separately declared variables.

Check Information

Group: Rule 06. Arrays (ARR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

ARR37-C

Introduced in R2019a

CERT C: Rule ARR38-C

Guarantee that library functions do not form invalid pointers

Description

Rule Definition

Guarantee that library functions do not form invalid pointers.

Polyspace Implementation

This checker checks for these issues:

- **Mismatch between data length and size**.
- **Invalid use of standard library memory routine**.
- **Possible misuse of sizeof**.
- **Buffer overflow from incorrect string format specifier**.
- **Invalid use of standard library string routine**.
- **Destination buffer overflow in string manipulation**.
- **Destination buffer underflow in string manipulation**.

Examples

Mismatch between data length and size

Issue

Mismatch between data length and size looks for memory copying functions such as `memcpy`, `memset`, or `memmove`. If you do not control the length argument and data buffer argument properly, Bug Finder raises a defect.

Risk

If an attacker can manipulate the data buffer or length argument, the attacker can cause buffer overflow by making the actual data size smaller than the length.

This mismatch in length allows the attacker to copy memory past the data buffer to a new location. If the extra memory contains sensitive information, the attacker can now access that data.

This defect is similar to the SSL Heartbleed bug.

Fix

When copying or manipulating memory, compute the length argument directly from the data so that the sizes match.

Example - Copy Buffer of Data

```
#include <stdlib.h>
#include <string.h>
```

```
typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    memcpy(&(beta.data[num]), os->data + 2, length);

    return(1);
}
```

This function copies the buffer `alpha` into a buffer `beta`. However, the `length` variable is not related to `data+2`.

Correction — Check Buffer Length

One possible correction is to check the length of your buffer against the maximum value minus 2. This check ensures that you have enough space to copy the data to the `beta` structure.

```
#include <stdlib.h>
#include <string.h>

typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    if (length<(os->max -2)) {
        memcpy(&(beta.data[num]), os->data + 2, length);
    }

    return(1);

}
```

Invalid use of standard library memory routine

Issue

Invalid use of standard library memory routine occurs when a memory library function is called with invalid arguments. For instance, the `memcpy` function copies to an array that cannot accommodate the number of bytes copied.

Risk

Use of a memory library function with invalid arguments can result in issues such as buffer overflow.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Invalid Use of Standard Library Memory Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string `str2` is 5, but six characters of string `str1` are copied into `str2` using the `memcpy` function.

Correction — Call Function with Valid Arguments

One possible correction is to adjust the size of `str2` so that it accommodates the characters copied with the `memcpy` function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];

  printf("Enter string:\n");
```

```
  scanf("%s",str1);

  memcpy(str2,str1,6);
  return str2;
 }
```

Possible misuse of sizeof

Issue

Possible misuse of sizeof occurs when Polyspace Bug Finder detects possibly unintended results from the use of `sizeof` operator. For instance:

- You use the `sizeof` operator on an array parameter name, expecting the array size. However, the array parameter name by itself is a pointer. The `sizeof` operator returns the size of that pointer.
- You use the `sizeof` operator on an array element, expecting the array size. However, the operator returns the size of the array element.
- The size argument of certain functions such as `strncmp` or `wcsncpy` is incorrect because you used the `sizeof` operator earlier with possibly incorrect expectations. For instance:
  - In a function call `strncmp(string1, string2, num)`, `num` is obtained from an incorrect use of the `sizeof` operator on a pointer.
  - In a function call `wcsncpy(destination, source, num)`, `num` is the not the number of wide characters but a size in bytes obtained by using the `sizeof` operator. For instance, you use `wcsncpy(destination, source, sizeof(destination) - 1)` instead of `wcsncpy(destination, source, (sizeof(desintation)/sizeof(wchar_t)) - 1)`.

Risk

Incorrect use of the `sizeof` operator can cause the following issues:

- If you expect the `sizeof` operator to return array size and use the return value to constrain a loop, the number of loop runs are smaller than what you expect.
- If you use the return value of `sizeof` operator to allocate a buffer, the buffer size is smaller than what you require. Insufficient buffer can lead to resultant weaknesses such as buffer overflows.
- If you use the return value of `sizeof` operator incorrectly in a function call, the function does not behave as you expect.

Fix

Possible fixes are:

- Do not use the `sizeof` operator on an array parameter name or array element to determine array size.

  The best practice is to pass the array size as a separate function parameter and use that parameter in the function body.
- Use the `sizeof` operator carefully to determine the number argument of functions such as `strncmp` or `wcsncpy`. For instance, for wide string functions such as `wcsncpy`, use the number of wide characters as argument instead of the number of bytes.

Example - `sizeof` Used Incorrectly to Determine Array Size

```
#define MAX_SIZE 1024
```

```
void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < sizeof(a)/sizeof(int); i++)    {
        a[i] = i + 1;
    }
}
```

In this example, `sizeof(a)` returns the size of the pointer `a` and not the array size.

Correction — Determine Array Size in Another Way

One possible correction is to use another means to determine the array size.

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < MAX_SIZE; i++)    {
        a[i] = i + 1;
    }
}
```

Buffer overflow from incorrect string format specifier

Issue

Buffer overflow from incorrect string format specifier occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

Risk

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

Fix

Use a format specifier that is compatible with the memory buffer size.

Example - Memory Buffer Overflow

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf);
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

Correction — Use Smaller Precision in Format Specifier

One possible correction is to use a smaller precision in the format specifier.

```
#include <stdio.h>

void func (char *str[]) {
```

```
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

Invalid use of standard library string routine

Issue

Invalid use of standard library string routine occurs when a string library function is called with invalid arguments.

Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Invalid Use of Standard Library String Routine Error

```
 #include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>
```

```
char* Copy_String(void)
{
 char *res;
 /*Fix: gbuffer has equal or larger size than text */
 char gbuffer[20],text[20]="ABCDEFGHIJKL";

 res=strcpy(gbuffer,text);

 return(res);
}
```

Destination buffer overflow in string manipulation

Issue

Destination buffer overflow in string manipulation occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

Example - Buffer Overflow in `sprintf` Use

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

Destination buffer underflow in string manipulation

Issue

Destination buffer underflow in string manipulation occurs when certain string manipulation functions write to their destination buffer argument at a negative offset from the beginning of the buffer.

For instance, for the function `sprintf(char* buffer, const char* format)`, you obtain the `buffer` from an operation `buffer = (char*)arr; ... buffer += offset;`. `arr` is an array and `offset` is a negative value.

Risk

Buffer underflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer underflow also introduces the risk of code injection.

Fix

If the destination buffer argument results from pointer arithmetic, see if you are decrementing a pointer. Fix the pointer decrement by modifying either the original value before decrement or the decrement value.

Example - Buffer Underflow in `sprintf` Use

```
#include <stdio.h>
#define offset -2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

In this example, `&buffer[offset]` is at a negative offset from the memory allocated to `buffer`.

Correction — Change Pointer Decrementer

One possible correction is to change the value of `offset`.

```
#include <stdio.h>
#define offset 2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

Check Information

Group: Rule 06. Arrays (ARR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

ARR38-C

Introduced in R2019a

CERT C: Rule ARR39-C

Do not add or subtract a scaled integer to a pointer

Description

Rule Definition

Do not add or subtract a scaled integer to a pointer.

Polyspace Implementation

This checker checks for **Incorrect pointer scaling**.

Examples

Incorrect pointer scaling

Issue

Incorrect pointer scaling occurs when Polyspace Bug Finder considers that you are ignoring the implicit scaling in pointer arithmetic.

For instance, the defect can occur in the following situations.

Situation	Risk	Possible Fix
You use the `sizeof` operator in arithmetic operations on a pointer.	The `sizeof` operator returns the size of a data type in number of bytes. Pointer arithmetic is already implicitly scaled by the size of the data type of the pointed variable. Therefore, the use of `sizeof` in pointer arithmetic produces unintended results.	Do not use `sizeof` operator in pointer arithmetic.
You perform arithmetic operations on a pointer, and then apply a cast.	Pointer arithmetic is implicitly scaled. If you do not consider this implicit scaling, casting the result of a pointer arithmetic produces unintended results.	Apply the cast before the pointer arithmetic.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Use of `sizeof` Operator

```
void func(void) {
    int arr[5] = {1,2,3,4,5};
    int *ptr = arr;

    int value_in_position_2 = *(ptr + 2*(sizeof(int)));
}
```

In this example, the operation `2*(sizeof(int))` returns twice the size of an `int` variable in bytes. However, because pointer arithmetic is implicitly scaled, the number of bytes by which `ptr` is offset is `2*(sizeof(int))*(sizeof(int))`.

In this example, the incorrect scaling shifts `ptr` outside the bounds of the array. Therefore, a **Pointer access out of bounds** error appears on the * operation.

Correction — Remove `sizeof` Operator

One possible correction is to remove the `sizeof` operator.

```
void func(void) {
    int arr[5] = {1,2,3,4,5};
    int *ptr = arr;

    int value_in_position_2 = *(ptr + 2);
}
```

Example - Cast Following Pointer Arithmetic

```
int func(void) {
    int x = 0;
    char r = *(char *)(&x + 1);
    return r;
}
```

In this example, the operation `&x + 1` offsets `&x` by `sizeof(int)`. Following the operation, the resulting pointer points outside the allowed buffer. When you dereference the pointer, a **Pointer access out of bounds** error appears on the * operation.

Correction — Apply Cast Before Pointer Arithmetic

If you want to access the second byte of `x`, first cast `&x` to a `char*` pointer and then perform the pointer arithmetic. The resulting pointer is offset by `sizeof(char)` bytes and still points within the allowed buffer, whose size is `sizeof(int)` bytes.

```
int func(void) {
    int x = 0;
    char r = *((char *)(&x )+ 1);
    return r;
}
```

Check Information

Group: Rule 06. Arrays (ARR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

ARR39-C

Introduced in R2019a

CERT C: Rule STR30-C

Do not attempt to modify string literals

Description

Rule Definition

Do not attempt to modify string literals.

Polyspace Implementation

This checker checks for **Writing to const qualified object**.

Examples

Writing to const qualified object

Issue

Writing to `const` qualified object occurs when you do one of the following:

- Use a `const`-qualified object as the destination of an assignment.
- Pass a `const`-qualified object to a function that modifies the argument.

For instance, the defect can occur in the following situations:

- You pass a `const`-qualified object as first argument of one of the following functions:
  - `mkstemp`
  - `mkostemp`
  - `mkostemps`
  - `mkdtemp`
- You pass a `const`-qualified object as the destination argument of one of the following functions:
  - `strcpy`
  - `strncpy`
  - `strcat`
  - `memset`
- You perform a write operation on a `const`-qualified object.

Risk

The risk depends upon the modifications made to the `const`-qualified object.

Situation	Risk
Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on.	These functions replace the last six characters of their first argument with a string. Therefore, they expect a modifiable `char` array as their first argument.
Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on.	These functions modify their destination argument. Therefore, they expect a modifiable `char` array as their destination argument.
Writing to the object	The `const` qualifier implies an agreement that the value of the object will not be modified. By writing to a `const`-qualified object, you break the agreement. The result of the operation is undefined.

Fix

The fix depends on the modification made to the `const`-qualified object.

Situation	Fix
Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on.	Pass a non-`const` object as first argument of the function.
Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on.	Pass a non-`const` object as destination argument of the function.
Writing to the object	Perform the write operation on a non-`const` object.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Writing to `const`-Qualified Object

```
#include <string.h>

const char* buffer = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

In this example, because `buffer` is `const`-qualified, `strchr(buffer,'X')` returns a `const`-qualified `char*` pointer. When this `char*` pointer is used as the destination argument of `strcpy`, a **Writing to const qualified object** error appears.

Correction — Copy `const`-Qualified Object to Non-`const` Object

One possible correction is to assign the constant string to a non-`const` object and use the non-`const` object as destination argument of `strchr`.

```
#include <string.h>
```

```
char buffer[] = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

Check Information

Group: Rule 07. Characters and Strings (STR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

STR30-C

Introduced in R2019a

CERT C: Rule STR31-C

Guarantee that storage for strings has sufficient space for character data and the null terminator

Description

Rule Definition

Guarantee that storage for strings has sufficient space for character data and the null terminator.

Polyspace Implementation

This checker checks for these issues:

- **Use of dangerous standard function**.
- **Missing null in string array**.
- **Buffer overflow from incorrect string format specifier**.
- **Destination buffer overflow in string manipulation**.

Examples

Use of dangerous standard function

Issue

The **Use of dangerous standard function** check highlights uses of functions that are inherently dangerous or potentially dangerous given certain circumstances. The following table lists possibly dangerous functions, the risks of using each function, and what function to use instead.

Dangerous Function	Risk Level	Safer Function
`gets`	Inherently dangerous — You cannot control the length of input from the console.	`fgets`
`cin`	Inherently dangerous — You cannot control the length of input from the console.	Avoid or prefaces calls to `cin` with `cin.width`.
`strcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`strncpy`
`stpcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`stpncpy`
`lstrcpy` or `StrCpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy`

Dangerous Function	Risk Level	Safer Function
`strcat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`strncat`, `strlcat`, or `strcat_s`
`lstrcat` or `StrCat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`StringCbCat`, `StringCchCat`, `strncay`, `strcat_s`, or `strlcat`
`wcpcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`wcpncpy`
`wcscat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`wcsncat`, `wcslcat`, or `wcncat_s`
`wcscpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`wcsncpy`
`sprintf`	Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur.	`snprintf`
`vsprintf`	Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur.	`vsnprintf`

Risk

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Using `sprintf`

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;
```

```
    if (sprintf(dst, "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

Correction — Use `snprintf` with Buffer Size

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

Missing null in string array

Issue

Missing null in string array occurs when a string does not have enough space to terminate with a null character `'\0'`.

This defect applies only for projects in C.

Risk

A buffer overflow can occur if you copy a string to an array without assuming the implicit null terminator.

Fix

If you initialize a character array with a literal, avoid specifying the array bounds.

```
char three[]  = "THREE";
```

The compiler automatically allocates space for a null terminator. In the preceding example, the compiler allocates sufficient space for five characters and a null terminator.

If the issue occurs after initialization, you might have to increase the size of the array by one to account for the null terminator.

In certain circumstances, you might want to initialize the character array with a sequence of characters instead of a string. In this situation, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Array size is too small

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[5] = "THREE";
}
```

The character array `three` has a size of 5 and 5 characters `'T'`, `'H'`, `'R'`, `'E'`, and `'E'`. There is no room for the null character at the end because `three` is only five bytes large.

Correction — Increase Array Size

One possible correction is to change the array size to allow for the five characters plus a null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[6] = "THREE";
}
```

Correction — Change Initialization Method

One possible correction is to initialize the string by leaving the array size blank. This initialization method allocates enough memory for the five characters and a terminating-null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[]  = "THREE";
}
```

Buffer overflow from incorrect string format specifier

Issue

Buffer overflow from incorrect string format specifier occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

Risk

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

Fix

Use a format specifier that is compatible with the memory buffer size.

Example - Memory Buffer Overflow

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf);
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

Correction — Use Smaller Precision in Format Specifier

One possible correction is to use a smaller precision in the format specifier.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

Destination buffer overflow in string manipulation

Issue

Destination buffer overflow in string manipulation occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

Example - Buffer Overflow in `sprintf` Use

```
#include <stdio.h>

void func(void) {
```

```
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

Check Information

Group: Rule 07. Characters and Strings (STR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

STR31-C

Introduced in R2019a

CERT C: Rule STR32-C

Do not pass a non-null-terminated character sequence to a library function that expects a string

Description

Rule Definition

Do not pass a non-null-terminated character sequence to a library function that expects a string.

Polyspace Implementation

This checker checks for these issues:

- **Invalid use of standard library string routine**.
- **Tainted NULL or non-null-terminated string**.

Examples

Invalid use of standard library string routine

Issue

Invalid use of standard library string routine occurs when a string library function is called with invalid arguments.

Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Invalid Use of Standard Library String Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_String(void)
```

```
{
 char *res;
 char gbuffer[5],text[20]="ABCDEFGHIJKL";

 res=strcpy(gbuffer,text);
 /* Error: Size of text is less than gbuffer */

 return(res);
}
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

Tainted NULL or non-null-terminated string

Issue

Tainted NULL or non-null-terminated string looks for strings from unsecure sources that are being used in string manipulation routines that implicitly dereference the string buffer. For example, `strcpy` or `sprintf`.

Tainted NULL or non-null-terminated string raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

Note If you reference a string using the form `ptr[i]`, `*ptr`, or pointer arithmetic, Bug Finder raises a **Use of tainted pointer** defect instead. The **Tainted NULL or non-null-terminated string** defect is raised only when the pointer is used as a string.

Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

Example - Getting String from Input Argument

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

Correction — Validate the Data

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

int sanitize_str(char* s) {
  int res = 0;
  if (s && (strlen(s) > 0)) { // TAINTED_STRING only flagged here
    // - string is not null
    // - string has a positive and limited size
    // - TAINTED_STRING on strlen used as a firewall
    res = 1;
  }
```

```
  return res;
}

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
      strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  if (sanitize_str(userstr))
    strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}
```

Correction — Validate the Data

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

Check Information

Group: Rule 07. Characters and Strings (STR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

STR32-C

Introduced in R2019a

CERT C: Rule STR34-C

Cast characters to unsigned char before converting to larger integer sizes

Description

Rule Definition

Cast characters to unsigned char before converting to larger integer sizes.

Polyspace Implementation

This checker checks for **Misuse of sign-extended character value**.

Examples

Misuse of sign-extended character value

Issue

Misuse of sign-extended character value occurs when you convert a signed or plain `char` data type to a wider integer data type with sign extension. You then use the resulting sign-extended value as array index, for comparison with EOF or as argument to a character-handling function.

Risk

Comparison with EOF: Suppose, your compiler implements the plain `char` type as signed. In this implementation, the character with the decimal form of 255 (–1 in two's complement form) is stored as a signed value. When you convert a `char` variable to the wider data type `int` for instance, the sign bit is preserved (sign extension). This sign extension results in the character with the decimal form 255 being converted to the integer –1, which cannot be distinguished from EOF.

Use as array index: By similar reasoning, you cannot use sign-extended plain `char` variables as array index. If the sign bit is preserved, the conversion from `char` to `int` can result in negative integers. You must use positive integer values for array index.

Argument to character-handling function: By similar reasoning, you cannot use sign-extended plain `char` variables as arguments to character-handling functions declared in `ctype.h`, for instance, `isalpha()` or `isdigit()`. According to the C11 standard (Section 7.4), if you supply an integer argument that cannot be represented as `unsigned char` or EOF, the resulting behavior is undefined.

Fix

Before conversion to a wider integer data type, cast the signed or plain `char` value explicitly to `unsigned char`.

Example - Sign-Extended Character Value Compared with EOF

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];
```

```
static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = *buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

In this example, the function `parser` can traverse a string input `buf`. If a character in the string has the decimal form 255, when converted to the `int` variable `c`, its value becomes –1, which is indistinguishable from `EOF`. The later comparison with `EOF` can lead to a false positive.

Correction — Cast to `unsigned char` Before Conversion

One possible correction is to cast the plain `char` value to `unsigned char` before conversion to the wider `int` type.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];

static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = (unsigned char)*buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

Check Information

Group: Rule 07. Characters and Strings (STR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics
“Check for Coding Standard Violations”

External Websites
STR34-C

Introduced in R2019a

CERT C: Rule STR37-C

Arguments to character-handling functions must be representable as an unsigned char

Description

Rule Definition

Arguments to character-handling functions must be representable as an unsigned char.

Polyspace Implementation

This checker checks for **Invalid use of standard library integer routine**.

Examples

Invalid use of standard library integer routine

Issue

Invalid use of standard library integer routine occurs when you use invalid arguments with an integer function from the standard library. This defect picks up:

- Character Conversion

  ```
  toupper, tolower
  ```
- Character Checks

  ```
  isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct,
  isspace, isupper, isxdigit
  ```
- Integer Division

  ```
  div, ldiv
  ```
- Absolute Values

  ```
  abs, labs
  ```

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Absolute Value of Large Negative

```
#include <limits.h>
#include <stdlib.h>
```

```
int absoluteValue(void) {

    int neg = INT_MIN;
    return abs(neg);
}
```

The input value to `abs` is `INT_MIN`. The absolute value of `INT_MIN` is `INT_MAX+1`. This number cannot be represented by the type `int`.

Correction — Change Input Argument

One possible correction is to change the input value to fit returned data type. In this example, change the input value to `INT_MIN+1`.

```
#include <limits.h>
#include <stdlib.h>

int absoluteValue(void) {

    int neg = INT_MIN+1;
    return abs(neg);
}
```

Check Information

Group: Rule 07. Characters and Strings (STR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

STR37-C

Introduced in R2019a

CERT C: Rule STR38-C

Do not confuse narrow and wide character strings and functions

Description

Rule Definition

Do not confuse narrow and wide character strings and functions.

Polyspace Implementation

This checker checks for **Misuse of narrow or wide character string**.

Examples

Misuse of narrow or wide character string

Issue

Misuse of narrow or wide character string occurs when you pass a narrow character string to a wide string function, or a wide character string to a narrow string function.

Misuse of narrow or wide character string raises no defect on operating systems where narrow and wide character strings have the same size.

Risk

Using a narrow character string with a wide string function, or vice versa, can result in unexpected or undefined behavior.

If you pass a wide character string to a narrow string function, you can encounter these issues:

- Data truncation. If the string contains null bytes, a copy operation using `strncpy()` can terminate early.
- Incorrect string length. `strlen()` returns the number of characters of a string up to the first null byte. A wide string can have additional characters after its first null byte.

If you pass a narrow character string to a wide string function, you can encounter this issue:

- Buffer overflow. In a copy operation using `wcsncpy()`, the destination string might have insufficient memory to store the result of the copy.

Fix

Use the narrow string functions with narrow character strings. Use the wide string functions with wide character strings.

Example - Passing Wide Character Strings to `strncpy()`

```
#include <string.h>
#include <wchar.h>

void func(void)
```

```
{
    wchar_t wide_str1[]  = L"0123456789";
    wchar_t wide_str2[] =  L"0000000000";
    strncpy(wide_str2, wide_str1, 10);
}
```

In this example, `strncpy()` copies 10 wide characters from `wide_strt1` to `wide_str2`. If `wide_str1` contains null bytes, the copy operation can end prematurely and truncate the wide character string.

Correction — Use `wcsncpy()` to Copy Wide Character Strings

One possible correction is to use `wcsncpy()` to copy `wide_str1` to `wide_str2`.

```
#include <string.h>
#include <wchar.h>

void func(void)
{
    wchar_t wide_str1[]  = L"0123456789";
    wchar_t wide_str2[] =  L"0000000000";
    wcsncpy(wide_str2, wide_str1, 10);
}
```

Check Information

Group: Rule 07. Characters and Strings (STR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

STR38-C

Introduced in R2019a

CERT C: Rule MEM30-C

Do not access freed memory

Description

Rule Definition

Do not access freed memory.

Polyspace Implementation

This checker checks for **Use of previously freed pointer**.

Examples

Use of previously freed pointer

Issue

Use of previously freed pointer occurs when you access a block of memory after freeing the block using the `free` function.

Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to access this block of memory can result in unpredictable behavior or even a segmentation fault.

Fix

The fix depends on the root cause of the defect. See if you intended to free the memory later or allocate another memory block to the pointer before access.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before dereferencing pointers, check them for NULL values and handle the error. In this way, you are protected against accessing a freed block.

Example - Use of Previously Freed Pointer Error

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   {
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);

    j = *pi + shift;
    /* Defect: Reading a freed pointer */
```

```
    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, dereferencing`pi` after the `free` statement is not valid.

Correction — Free Pointer After Use

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
#include <stdlib.h>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

Check Information

Group: Rule 08. Memory Management (MEM)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MEM30-C

Introduced in R2019a

CERT C: Rule MEM31-C

Free dynamically allocated memory when no longer needed

Description

Rule Definition

Free dynamically allocated memory when no longer needed.

Polyspace Implementation

This checker checks for **Memory leak**.

Examples

Memory leak

Issue

Memory leak occurs when you do not free a block of memory allocated through `malloc`, `calloc`, `realloc`, or `new`. If the memory is allocated in a function, the defect does not occur if:

- Within the function, you free the memory using `free` or `delete`.
- The function returns the pointer assigned by `malloc`, `calloc`, `realloc`, or `new`.
- The function stores the pointer in a global variable or in a parameter.

Risk

Dynamic memory allocation functions such as `malloc` allocate memory on the heap. If you do not release the memory after use, you reduce the amount of memory available for another allocation. On embedded systems with limited memory, you might end up exhausting available heap memory even during program execution.

Fix

Determine the scope where the dynamically allocated memory is accessed. Free the memory block at the end of this scope.

To free a block of memory, use the `free` function on the pointer that was used during memory allocation. For instance:

```
ptr = (int*)malloc(sizeof(int));
...
free(ptr);
```

It is a good practice to allocate and free memory in the same module at the same level of abstraction. For instance, in this example, `func` allocates and frees memory at the same level but `func2` does not.

```
void func() {
  ptr = (int*)malloc(sizeof(int));
  {
    ...
```

```
  }
  free(ptr);
}

void func2() {
  {
   ptr = (int*)malloc(sizeof(int));
   ...
  }
  free(ptr);
}
```

See CERT-C Rule MEM00-C.

Example - Dynamic Memory Not Released Before End of Function

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }


    *pi = 42;
    /* Defect: pi is not freed */
}
```

In this example, `pi` is dynamically allocated by `malloc`. The function `assign_memory` does not free the memory, nor does it return `pi`.

Correction — Free Memory

One possible correction is to free the memory referenced by `pi` using the `free` function. The `free` function must be called before the function `assign_memory` terminates

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }
    *pi = 42;

    /* Fix: Free the pointer pi*/
    free(pi);
}
```

Correction — Return Pointer from Dynamic Allocation

Another possible correction is to return the pointer `pi`. Returning `pi` allows the function calling `assign_memory` to free the memory block using `pi`.

```
#include<stdlib.h>
#include<stdio.h>

int* assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
            printf("Memory allocation failed");
            return(pi);
        }
    *pi = 42;

    /* Fix: Return the pointer pi*/
    return(pi);
}
```

Example - Memory Leak with New/Delete

```
#define NULL '\0'

void initialize_arr1(void)
{
    int *p_scalar = new int(5);
}

void initialize_arr2(void)
{
    int *p_array = new int[5];
}
```

In this example, the functions create two variables, `p_scalar` and `p_array`, using the `new` keyword. However, the functions end without cleaning up the memory for these pointers. Because the functions used `new` to create these variables, you must clean up their memory by calling `delete` at the end of each function.

Correction — Add Delete

To correct this error, add a `delete` statement for every `new` initialization. If you used brackets `[]` to instantiate a variable, you must call delete with brackets as well.

```
#define NULL '\0'

void initialize_arrs(void)
{
    int *p_scalar = new int(5);
    int *p_array = new int[5];

    delete p_scalar;
    p_scalar = NULL;
```

```
    delete[] p_array;
    p_scalar = NULL;
}
```

Check Information

Group: Rule 08. Memory Management (MEM)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MEM31-C

Introduced in R2019a

CERT C: Rule MEM33-C

Allocate and copy structures containing a flexible array member dynamically

Description

Rule Definition

Allocate and copy structures containing a flexible array member dynamically.

Polyspace Implementation

This checker checks for **Misuse of structure with flexible array member**.

Examples

Misuse of structure with flexible array member

Issue

Misuse of structure with flexible array member occurs when:

- You define an object with a flexible array member of unknown size at compilation time.
- You make an assignment between structures with a flexible array member without using `memcpy()` or a similar function.
- You use a structure with a flexible array member as an argument to a function and pass the argument by value.
- Your function returns a structure with a flexible array member.

A flexible array member has no array size specified and is the last element of a structure with at least two named members.

Risk

If the size of the flexible array member is not defined, it is ignored when allocating memory for the containing structure. Accessing such a structure has undefined behavior.

Fix

- Use `malloc()` or a similar function to allocate memory for a structure with a flexible array member.
- Use `memcpy()` or a similar function to copy a structure with a flexible array member.
- Pass a structure with a flexible array member as a function argument by pointer.

Example - Structure Passed By Value to Function

```
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
```

```
struct example_struct
{
    size_t num;
    int data[];
};

extern void arg_by_value(struct example_struct s);

void func(void)
{
    struct example_struct *flex_struct;
    size_t i;
    size_t array_size = 4;
    /* Dynamically allocate memory for the struct */
    flex_struct = (struct example_struct *)
        malloc(sizeof(struct example_struct) + sizeof(int) * array_size);
    if (flex_struct == NULL)
    {
        /* Handle error */
    }
    /* Initialize structure */
    flex_struct->num = array_size;
    for (i = 0; i < array_size; ++i)
    {
        flex_struct->data[i] = 0;
    }
    /* Handle structure */

    /* Argument passed by value. 'data' not
    copied to passed value. */
    arg_by_value(*flex_struct);

    /* Free dynamically allocated memory */
    free(flex_struct);
}
```

In this example, `flex_struct` is passed by value as an argument to `arg_by_value`. As a result, the flexible array member data is not copied to the passed argument.

Correction — Pass Structure by Pointer to Function

To ensure that all the members of the structure are copied to the passed argument, pass `flex_struct` to `arg_by_pointer` by pointer.

```
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>


struct example_struct
{
    size_t num;
    int data[];
```

```
};

extern void arg_by_pointer(struct example_struct *s);

void func(void)
{
    struct example_struct *flex_struct;
    size_t i;
    size_t array_size = 4;
    /* Dynamically allocate memory for the struct */
    flex_struct = (struct example_struct *)
        malloc(sizeof(struct example_struct) + sizeof(int) * array_size);
    if (flex_struct == NULL)
    {
        /* Handler error */
    }
    /* Initialize structure */
    flex_struct->num = array_size;
    for (i = 0; i < array_size; ++i)
    {
        flex_struct->data[i] = 0;
    }
    /* Handle structure */

    /* Structure passed by pointer */
    arg_by_pointer(flex_struct);

    /* Free dynamically allocated memory */
    free(flex_struct);
}
```

Check Information

Group: Rule 08. Memory Management (MEM)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MEM33-C

Introduced in R2019a

CERT C: Rule MEM34-C

Only free memory allocated dynamically

Description

Rule Definition

Only free memory allocated dynamically.

Polyspace Implementation

This checker checks for **Invalid free of pointer**.

Examples

Invalid free of pointer

Issue

Invalid free of pointer occurs when a block of memory released using the `free` function was not previously allocated using `malloc`, `calloc`, or `realloc`.

Risk

The `free` function releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.

The issue can highlight coding errors. For instance, you perhaps wanted to use the `free` function or a previous `malloc` function on a different pointer.

Fix

In most cases, you can fix the issue by removing the `free` statement. If the pointer is not allocated memory from the heap with `malloc` or `calloc`, you do not need to free the pointer. You can simply reuse the pointer as required.

If the issue highlights a coding error such as use of `free` or `malloc` on the wrong pointer, correct the error.

If the issue occurs because you use the `free` function to free memory allocated with the `new` operator, replace the `free` function with the `delete` operator.

Example - Invalid Free of Pointer Error

```
#include <stdlib.h>

void Assign_Ones(void)
{
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;

  free(p);
```

```
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer `p` is deallocated using the `free` function. However, `p` points to a memory location that was not dynamically allocated.

Correction — Remove Pointer Deallocation

If the number of elements of the array `p` is known at compile time, one possible correction is to remove the deallocation of the pointer `p`.

```
#include <stdlib.h>

void Assign_Ones(void)
 {
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;
  /* Fix: Remove deallocation of p */
 }
```

Correction — Introduce Pointer Allocation

If the number of elements of the array `p` is not known at compile time, one possible correction is to dynamically allocate memory to the array `p`.

```
#include <stdlib.h>

void Assign_Ones(int num)
{
  int *p;
  /* Fix: Allocate memory dynamically to p */
  p=(int*) calloc(10,sizeof(int));
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
}
```

Check Information

Group: Rule 08. Memory Management (MEM)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MEM34-C

Introduced in R2019a

CERT C: Rule MEM35-C

Allocate sufficient memory for an object

Description

Rule Definition

Allocate sufficient memory for an object.

Polyspace Implementation

This checker checks for these issues:

- **Pointer access out of bounds**.
- **Memory allocation with tainted size**.

Examples

Pointer access out of bounds

Issue

Pointer access out of bounds occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

Memory allocation with tainted size

Issue

Memory allocation with tainted size checks memory allocation functions, such as `calloc` or `malloc`, for size arguments from unsecured sources.

Risk

Uncontrolled memory allocation can cause your program to request too much system memory. This consequence can lead to a crash due to an out-of-memory condition, or assigning too many resources.

Fix

Before allocating memory, check the value of your arguments to check that they do not exceed the bounds.

Example - Allocate Memory Using Input Argument

```
#include "stdlib.h"

int* bug_taintedmemoryallocsize(size_t size) {
    int* p = (int*)malloc(size);
    return p;
}
```

In this example, `malloc` allocates `size` amount of memory for the pointer `p`. `size` is an outside variable, so could be any size value. If the size is larger than the amount of memory you have available, your program could crash.

Correction — Check Size of Memory to be Allocated

One possible correction is to check the size of the memory that you want to allocate before performing the `malloc` operation. This example checks to see if the size is positive and less than the maximum size.

```
#include "stdlib.h"

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int* corrected_taintedmemoryallocsize(int size) {
    int* p = NULL;
    if (size>0 && size<SIZE128) {          /* Fix: Check entry range before use */
        p = (int*)malloc((unsigned int)size);
    }
    return p;
}
```

Check Information

Group: Rule 08. Memory Management (MEM)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MEM35-C

Introduced in R2019a

CERT C: Rule MEM36-C

Do not modify the alignment of objects by calling realloc()

Description

Rule Definition

Do not modify the alignment of objects by calling realloc().

Polyspace Implementation

This checker checks for **Alignment changed after memory reallocation**.

Examples

Alignment changed after memory reallocation

Issue

Alignment changed after memory reallocation occurs when you use `realloc()` to modify the size of objects with strict memory alignment requirements.

Risk

The pointer returned by `realloc()` can be suitably assigned to objects with less strict alignment requirements. A misaligned memory allocation can lead to buffer underflow or overflow, an illegally dereferenced pointer, or access to arbitrary memory locations. In processors that support misaligned memory, the allocation impacts the performance of the system.

Fix

To reallocate memory:

1. Resize the memory block.
   - In Windows, use `_aligned_realloc()` with the alignment argument used in `_aligned_malloc()` to allocate the original memory block.
   - In UNIX/Linux, use the same function with the same alignment argument used to allocate the original memory block.
2. Copy the original content to the new memory block.
3. Free the original memory block.

Note This fix has implementation-defined behavior. The implementation might not support the requested memory alignment and can have additional constraints for the size of the new memory.

Example - Memory Reallocated Without Preserving the Original Alignment

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define SIZE1024 1024

void func(void)
{
    size_t resize = SIZE1024;
    size_t alignment = 1 << 12; /* 4096 bytes alignment */
    int *ptr = NULL;
    int *ptr1;

    /* Allocate memory with 4096 bytes alignment */

    if (posix_memalign((void **)&ptr, alignment, sizeof(int)) != 0)
    {
        /* Handle error */
      }

    /*Reallocate memory without using the original alignment.
    ptr1 may not be 4096 bytes aligned. */

    ptr1 = (int *)realloc(ptr, sizeof(int) * resize);

    if (ptr1 == NULL)
    {
        /* Handle error */
    }

    /* Processing using ptr1 */

    /* Free before exit */
    free(ptr1);
}
```

In this example, the allocated memory is 4096-bytes aligned. `realloc()` then resizes the allocated memory. The new pointer `ptr1` might not be 4096-bytes aligned.

Correction — Specify the Alignment for the Reallocated Memory

When you reallocate the memory, use `posix_memalign()` and pass the alignment argument that you used to allocate the original memory.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE1024 1024

void func(void)
{
    size_t resize = SIZE1024;
    size_t alignment = 1 << 12; /* 4096 bytes alignment */
    int *ptr = NULL;

    /* Allocate memory with 4096 bytes alignment */
    if (posix_memalign((void **)&ptr, alignment, sizeof(int)) != 0)
    {
        /* Handle error */
```

```
    }

    /* Reallocate memory using the original alignment. */
    if (posix_memalign((void **)&ptr, alignment, sizeof(int) * resize) != 0)
    {
        /* Handle error */
        free(ptr);
        ptr = NULL;
    }

    /* Processing using ptr */

    /* Free before exit */
    free(ptr);
}
```

Check Information

Group: Rule 08. Memory Management (MEM)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MEM36-C

Introduced in R2019a

CERT C: Rule FIO30-C

Exclude user input from format strings

Description

Rule Definition

Exclude user input from format strings.

Polyspace Implementation

This checker checks for **Tainted string format**.

Examples

Tainted string format

Issue

Tainted string format detects string formatting with `printf`-style functions that contain elements from unsecure sources.

Risk

If you use externally controlled elements to format a string, you can cause buffer overflow or data-representation problems. An attacker can use these string formatting elements to view the contents of a stack using `%x` or write to a stack using `%n`.

Fix

Pass a static string to format string functions. This fix ensures that an external actor cannot control the string.

Another possible fix is to allow only the expected number of arguments. If possible, use functions that do not support the vulnerable `%n` operator in format strings.

Example - Get Elements from User Input

```
#include "stdio.h"

void taintedstringformat(char* userstr) {
    printf(userstr);
}
```

This example prints the input argument `userstr`. The string is unknown. If it contains elements such as `%`, `printf` can interpret `userstr` as a string format instead of a string, causing your program to crash.

Correction — Print as String

One possible correction is to print `userstr` explicitly as a string so that there is no ambiguity.

```
#include "stdio.h"
```

```
void taintedstringformat(char* userstr) {
    printf("%.20s", userstr);
}
```

Check Information

Group: Rule 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FIO30-C

Introduced in R2019a

CERT C: Rule FIO32-C

Do not perform operations on devices that are only appropriate for files

Description

Rule Definition

Do not perform operations on devices that are only appropriate for files.

Polyspace Implementation

This checker checks for **Inappropriate I/O operation on device files**.

Examples

Inappropriate I/O operation on device files

Issue

Inappropriate I/O operation on device files occurs when you do not check whether a file name parameter refers to a device file before you pass it to these functions:

- `fopen()`
- `fopen_s()`
- `freopen()`
- `remove()`
- `rename()`
- `CreateFile()`
- `CreateFileA()`
- `CreateFileW()`
- `_wfopen()`
- `_wfopen_s()`

Device files are files in a file system that provide an interface to device drivers. You can use these files to interact with devices.

Inappropriate I/O operation on device files does not raise a defect when:

- You use `stat` or `lstat`-family functions to check the file name parameter before calling the previously listed functions.
- You use a string comparison function to compare the file name against a list of device file names.

Risk

Operations appropriate only for regular files but performed on device files can result in denial-of-service attacks, other security vulnerabilities, or system failures.

Fix

Before you perform an I/O operation on a file:

- Use `stat()`, `lstat()`, or an equivalent function to check whether the file name parameter refers to a regular file.
- Use a string comparison function to compare the file name against a list of device file names.

Example - Using `fopen()` Without Checking `file_name`

```
#include <stdio.h>
#include <string.h>

#define SIZE1024 1024

FILE* func()
{

    FILE* f;
    const char file_name[SIZE1024] = "./tmp/file";

    if ((f = fopen(file_name, "w")) == NULL) {
        /*handle error */
    };
    /*operate on file */
}
```

In this example, `func()` operates on the file `file_name` without checking whether it is a regular file. If `file_name` is a device file, attempts to access it can result in a system failure.

Correction — Check File with `lstat()` Before Calling `fopen()`

One possible correction is to use `lstat()` and the `S_ISREG` macro to check whether the file is a regular file. This solution contains a `TOCTOU` race condition that can allow an attacker to modify the file after you check it but before the call to fopen(). To prevent this vulnerability, ensure that `file_name` refers to a file in a secure folder.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>

#define SIZE1024 1024

FILE* func()
{

    FILE* f;
    const char file_name[SIZE1024] = "./tmp/file";
    struct stat orig_st;
    if ((lstat(file_name, &orig_st) != 0) ||
        (!S_ISREG(orig_st.st_mode))) {
        exit(0);
    }
    if ((f = fopen(file_name, "w")) == NULL) {
        /*handle error */
    };
```

```
    /*operate on file */
}
```

Check Information

Group: Rule 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FIO32-C

Introduced in R2019a

CERT C: Rule FIO34-C

Distinguish between characters read from a file and EOF or WEOF

Description

Rule Definition

Distinguish between characters read from a file and EOF or WEOF.

Polyspace Implementation

This checker checks for **Character value absorbed into EOF**.

Examples

Character value absorbed into EOF

Issue

Character value absorbed into EOF occurs when you perform a data type conversion that makes a valid character value indistinguishable from `EOF` (End-of-File). Bug Finder flags the defect in one of the following situations:

- *End-of-File*: You perform a data type conversion such as from `int` to `char` that converts a non-EOF character value into `EOF`.

  ```
  char ch = (char)getchar()
  ```

  You then compare the result with EOF.

  ```
  if((int)ch == EOF)
  ```

  The conversion can be explicit or implicit.
- *Wide End-of-File*: You perform a data type conversion that can convert a non-WEOF wide character value into WEOF, and then compare the result with WEOF.

Risk

The data type `char` cannot hold the value `EOF` that indicates the end of a file. Functions such as `getchar` have return type `int` to accommodate `EOF`. If you convert from `int` to `char`, the values `UCHAR_MAX` (a valid character value) and `EOF` get converted to the same value -1 and become indistinguishable from each other. When you compare the result of this conversion with `EOF`, the comparison can lead to false detection of `EOF`. This rationale also applies to wide character values and `WEOF`.

Fix

Perform the comparison with `EOF` or `WEOF` before conversion.

Example - Return Value of `getchar` Converted to `char`

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define fatal_error() abort()

char func(void)
{
    char ch;
    ch = getchar();
    if (EOF == (int)ch) {
        fatal_error();
    }
    return ch;
}
```

In this example, the return value of `getchar` is implicitly converted to `char`. If `getchar` returns `UCHAR_MAX`, it is converted to -1, which is indistinguishable from EOF. When you compare with EOF later, it can lead to a false positive.

Correction — Perform Comparison with EOF Before Conversion

One possible correction is to first perform the comparison with EOF, and then convert from `int` to `char`.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

char func(void)
{
    int i;
    i = getchar();
    if (EOF == i) {
        fatal_error();
    }
    else {
        return (char)i;
    }
}
```

Check Information

Group: Rule 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FIO34-C

Introduced in R2019a

CERT C: Rule FIO37-C

Do not assume that fgets() or fgetws() returns a nonempty string when successful

Description

Rule Definition

Do not assume that fgets() or fgetws() returns a nonempty string when successful.

Polyspace Implementation

This checker checks for **Use of indeterminate string**.

Examples

Use of indeterminate string

Issue

Use of indeterminate string occurs when you do not check the validity of the buffer returned from `fgets`-family functions. The checker raises a defect when such a buffer is used as:

- An argument in standard functions that print or manipulate strings or wide strings.
- A return value.
- An argument in external functions with parameter type `const char *` or `const wchar_t *`.

Risk

If an `fgets`-family function fails, the content of its output buffer is indeterminate. Use of such a buffer has undefined behavior and can result in a program that stops working or other security vulnerabilities.

Fix

Reset the output buffer of an `fgets`-family function to a known string value when the function fails.

Example - Output of `fgets()` Passed to External Function

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func(void) {
    char buf[SIZE20];

    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
```

```
        /* 'buf' may contain an indeterminate string.  */
        ;
    }
    /* 'buf passed to external function */
    display_text(buf);
}
```

In this example, the output `buf` is passed to the external function `display_text()`, but its value is not reset if `fgets()` fails.

Correction — Reset `fgets()` Output on Failure

If `fgets()` fails, reset `buf` to a known value before you pass it to an external function.

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func1(void) {
    char buf[SIZE20];
    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* value of 'buf' reset after fgets() failure. */
        buf[0] = '\0';
    }
    /* 'buf' passed to external function */
    display_text(buf);
}
```

Check Information

Group: Rule 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FIO37-C

Introduced in R2019a

CERT C: Rule FIO38-C

Do not copy a FILE object

Description

Rule Definition

Do not copy a FILE object.

Polyspace Implementation

This checker checks for **Misuse of a FILE object**.

Examples

Misuse of a FILE object

Issue

Misuse of a FILE object occurs when:

- You dereference a pointer to a FILE object, including indirect dereference by using `memcmp()`.
- You modify an entire FILE object or one of its components through its pointer.
- You take the address of FILE object that was not returned from a call to an `fopen`-family function. No defect is raised if a macro defines the pointer as the address of a built-in FILE object, such as `#define ptr (&__stdout)`.

Risk

In some implementations, the address of the pointer to a FILE object used to control a stream is significant. A pointer to a copy of a FILE object is interpreted differently than a pointer to the original object, and can potentially result in operations on the wrong stream. Therefore, the use of a copy of a FILE object can cause the software to stop responding, which an attacker might exploit in denial-of-service attacks.

Fix

Do not make a copy of a FILE object. Do not use the address of a FILE object that was not returned from a successful call to an `fopen`-family function.

Example - Copy of FILE Object Used in `fputs()`

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

void fatal_error(void);

int func(void)
{
```

```
    /*'stdout' dereferenced and contents
        copied to 'my_stdout'. */
    FILE my_stdout = *stdout;

    /* Address of 'my_stdout' may not point to correct stream. */
    if (fputs("Hello, World!\n", &my_stdout) == EOF)
    {
        /* Handler error */
        fatal_error();
    }
    return 0;
}
```

In this example, FILE object `stdout` is dereferenced and its contents are copied to `my_stdout`. The contents of `stdout` might not be significant. `fputs()` is then called with the address of `my_stdout` as an argument. Because no call to `fopen()` or a similar function was made, the address of `my_stdout` might not point to the correct stream.

Correction — Copy the FILE Object Pointer

Declare `my_stdout` to point to the same address as `stdout` to ensure that you write to the correct stream when you call `fputs()`.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

void fatal_error(void);

int func(void)
{
    /* 'my_stdout' and 'stdout' point to the same object. */
    FILE *my_stdout = stdout;
    if (fputs("Hello, World!\n", my_stdout) == EOF)
    {
        /* Handler error */
        fatal_error();
    }
    return 0;
}
```

Check Information

Group: Rule 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FIO38-C

Introduced in R2019a

CERT C: Rule FIO39-C

Do not alternately input and output from a stream without an intervening flush or positioning call

Description

Rule Definition

Do not alternately input and output from a stream without an intervening flush or positioning call.

Polyspace Implementation

This checker checks for **Alternating input and output from a stream without flush or positioning call**.

Examples

Alternating input and output from a stream without flush or positioning call

Issue

Alternating input and output from a stream without flush or positioning call occurs when:

- You do not perform a flush or function positioning call between an output operation and a following input operation on a file stream in update mode.
- You do not perform a function positioning call between an input operation and a following output operation on a file stream in update mode.

Risk

Alternating input and output operations on a stream without an intervening flush or positioning call is undefined behavior.

Fix

Call `fflush()` or a file positioning function such as `fseek()` or `fsetpos()` between output and input operations on an update stream.

Call a file positioning function between input and output operations on an update stream.

Example - Read After Write Without Intervening Flush

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";

void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;
```

```
    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Read operation after write without
    intervening flush. */
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
          (void)fclose(file);
          /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

In this example, the file `demo.txt` is opened for reading and appending. After the call to `fwrite()`, a call to `fread()` without an intervening flush operation is undefined behavior.

Correction — Call `fflush()` Before the Read Operation

After writing data to the file, before calling `fread()`, perform a flush call.

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";


void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;

    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
```

```
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Buffer flush after write and before read */
    if (fflush(file) != 0)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

Check Information

Group: Rule 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FIO39-C

Introduced in R2019a

CERT C: Rule FIO40-C

Reset strings on fgets() or fgetws() failure

Description

Rule Definition

Reset strings on fgets() or fgetws() failure.

Polyspace Implementation

This checker checks for **Use of indeterminate string**.

Examples

Use of indeterminate string

Issue

Use of indeterminate string occurs when you do not check the validity of the buffer returned from `fgets`-family functions. The checker raises a defect when such a buffer is used as:

- An argument in standard functions that print or manipulate strings or wide strings.
- A return value.
- An argument in external functions with parameter type `const char *` or `const wchar_t *`.

Risk

If an `fgets`-family function fails, the content of its output buffer is indeterminate. Use of such a buffer has undefined behavior and can result in a program that stops working or other security vulnerabilities.

Fix

Reset the output buffer of an `fgets`-family function to a known string value when the function fails.

Example - Output of `fgets()` Passed to External Function

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func(void) {
    char buf[SIZE20];

    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
```

```
        /* 'buf' may contain an indeterminate string.  */
        ;
    }
    /* 'buf passed to external function */
    display_text(buf);
}
```

In this example, the output `buf` is passed to the external function `display_text()`, but its value is not reset if `fgets()` fails.

Correction — Reset `fgets()` Output on Failure

If `fgets()` fails, reset `buf` to a known value before you pass it to an external function.

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func1(void) {
    char buf[SIZE20];
    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* value of 'buf' reset after fgets() failure. */
        buf[0] = '\0';
    }
    /* 'buf' passed to external function */
    display_text(buf);
}
```

Check Information

Group: Rule 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FIO40-C

Introduced in R2019a

CERT C: Rule FIO41-C

Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects

Description

Rule Definition

Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects.

Polyspace Implementation

This checker checks for **Stream argument with possibly unintended side effects**.

Examples

Stream argument with possibly unintended side effects

Issue

Stream argument with possibly unintended side effects occurs when you call `getc()`, `putc()`, `getwc()`, or `putwc()` with a stream argument that has side effects.

Stream argument with possibly unintended side effects considers the following as stream side effects:

- Any assignment of a variable of a stream, such as `FILE *`, or any assignment of a variable of a deeper stream type, such as an array of `FILE *`.
- Any call to a function that manipulates a stream or a deeper stream type.

The number of defects raised corresponds to the number of side effects detected. When a stream argument is evaluated multiple times in a function implemented as a macro, a defect is raised for each evaluation that has a side effect.

A defect is also raised on functions that are not implemented as macros but that can be implemented as macros on another operating system.

Risk

If the function is implemented as an unsafe macro, the stream argument can be evaluated more than once, and the stream side effect happens multiple times. For instance, a stream argument calling `fopen()` might open the same file multiple times, which is unspecified behavior.

Fix

To ensure that the side effect of a stream happens only once, use a separate statement for the stream argument.

Example - Stream Argument of `getc()` Has Side Effect `fopen()`

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#define fatal_error() abort()

const char* myfile = "my_file.log";

void func(void)
{
    int c;
    FILE* fptr;
    /* getc() has stream argument fptr with
    * 2 side effects: call to fopen(), and assignment
    * of fptr
    */
    c = getc(fptr = fopen(myfile, "r"));
    if (c == EOF) {
        /* Handle error */
        (void)fclose(fptr);
        fatal_error();
    }
    if (fclose(fptr) == EOF) {
        /* Handle error */
        fatal_error();
    }
}

void main(void)
{
    func();

}
```

In this example, `getc()` is called with stream argument `fptr`. The stream argument has two side effects: the call to `fopen()` and the assignment of `fptr`. If `getc()` is implemented as an unsafe macro, the side effects happen multiple times.

Correction — Use Separate Statement for `fopen()`

One possible correction is to use a separate statement for `fopen()`. The call to `fopen()` and the assignment of `fptr` happen in this statement so there are no side effects when you pass `fptr` to `getc()`.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

#define fatal_error() abort()


const char* myfile = "my_file.log";

void func(void)
{
    int c;
    FILE* fptr;

    /* Separate statement for fopen()
    * before call to getc()
    */
    fptr = fopen(myfile, "r");
```

```
    if (fptr == NULL) {
        /* Handle error */
        fatal_error();
    }
    c = getc(fptr);
    if (c == EOF) {
        /* Handle error */
        (void)fclose(fptr);
        fatal_error();
    }
    if (fclose(fptr) == EOF) {
        /* Handle error */
        fatal_error();
    }
}

void main(void)
{
    func();

}
```

Check Information

Group: Rule 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FIO41-C

Introduced in R2019a

CERT C: Rule FIO42-C

Close files when they are no longer needed

Description

Rule Definition

Close files when they are no longer needed.

Polyspace Implementation

This checker checks for **Resource leak**.

Examples

Resource leak

Issue

Resource leak occurs when you open a file stream by using a `FILE` pointer but do not close it before:

- The end of the pointer's scope.
- Assigning the pointer to another stream.

Risk

If you do not release file handles explicitly as soon as possible, a failure can occur due to exhaustion of resources.

Fix

Close a `FILE` pointer before the end of its scope, or before you assign the pointer to another stream.

Example - `FILE` Pointer Not Released Before End of Scope

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

In this example, the file pointer `fp1` is pointing to a file `data1.txt`. Before `fp1` is explicitly dissociated from the file stream of `data1.txt`, it is used to access another file `data2.txt`.

Correction — Release `FILE` Pointer

One possible correction is to explicitly dissociate `fp1` from the file stream of `data1.txt`.

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );
    fclose(fp1);

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

Check Information

Group: Rule 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FIO42-C

Introduced in R2019a

CERT C: Rule FIO44-C

Only use values for fsetpos() that are returned from fgetpos()

Description

Rule Definition

Only use values for fsetpos() that are returned from fgetpos().

Polyspace Implementation

This checker checks for **Invalid file position**.

Examples

Invalid file position

Issue

Invalid file position occurs when the file position argument of `fsetpos()` uses a value that is not obtained from `fgetpos()`.

Risk

The function `fgetpos(FILE *stream, fpos_t *pos)` gets the current file position of the stream. When you use any other value as the file position argument of `fsetpos(FILE *stream, const fpos_t *pos)`, you might access an unintended location in the stream.

Fix

Use the value returned from a successful call to `fgetpos()` as the file position argument of `fsetpos()`.

Example - `memset()` Sets File Position Argument

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


FILE *func(FILE *file)
{
    fpos_t offset;
    if (file == NULL)
    {
        /* Handle error */
    }
    /* Store initial position in variable 'offset' */
    (void)memset(&offset, 0, sizeof(offset));

    /* Read data from file */

    /* Return to the initial position. offset was not
    returned from a call to fgetpos()    */
```

```
    if (fsetpos(file, &offset) != 0)
    {
        /* Handle error */
    }
    return file;
}
```

In this example, `fsetpos()` uses `offset` as its file position argument. However, the value of `offset` is set by `memset()`. The preceding code might access the wrong location in the stream.

Correction — Use a File Position Returned From `fgetpos()`

Call `fgetpos()`, and if it returns successfully, use the position argument in your call to `fsetpos()`.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

FILE *func(FILE *file)
{
    fpos_t offset;
    if (file == NULL)
    {
        /* Handle error */
    }
    /* Store initial position in variable 'offset'
    using fgetpos() */
    if (fgetpos(file, &offset) != 0)
    {
        /* Handle error */
    }

    /* Read data from file */

    /* Back to the initial position */
    if (fsetpos(file, &offset) != 0)
    {
        /* Handle error */
    }
    return file;
}
```

Check Information

Group: Rule 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FIO44-C

Introduced in R2019a

CERT C: Rule FIO45-C

Avoid TOCTOU race conditions while accessing files

Description

Rule Definition

Avoid TOCTOU race conditions while accessing files.

Polyspace Implementation

This checker checks for **File access between time of check and use (TOCTOU)**.

Examples

File access between time of check and use (TOCTOU)

Issue

File access between time of check and use (TOCTOU) detects race condition issues between checking the existence of a file or folder, and using a file or folder.

Risk

An attacker can access and manipulate your file between your check for the file and your use of a file. Symbolic links are particularly risky because an attacker can change where your symbolic link points.

Fix

Before using a file, do not check its status. Instead, use the file and check the results afterward.

Example - Check File Before Using

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern void print_tofile(FILE* f);

void toctou(char * log_path) {
    if (access(log_path, W_OK)==0) {
        FILE* f = fopen(log_path, "w");
        if (f) {
            print_tofile(f);
            fclose(f);
        }
    }
}
```

In this example, before opening and using the file, the function checks if the file exists. However, an attacker can change the file between the first and second lines of the function.

Correction — Open Then Check

One possible correction is to open the file, and then check the existence and contents afterward.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern void print_tofile(FILE* f);

void toctou(char * log_path) {
    int fd = open(log_path, O_WRONLY);
    if (fd!=-1) {
        FILE *f = fdopen(fd, "w");
        if (f) {
            print_tofile(f);
            fclose(f);
        }
    }
}
```

Check Information

Group: Rule 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FIO45-C

Introduced in R2019a

CERT C: Rule FIO46-C

Do not access a closed file

Description

Rule Definition

Do not access a closed file.

Polyspace Implementation

This checker checks for **Use of previously closed resource**.

Examples

Use of previously closed resource

Issue

Use of previously closed resource occurs when a function operates on a stream that you closed earlier in your code.

Risk

The standard states that the value of a `FILE*` pointer is indeterminate after you close the stream associated with it. Operations using the `FILE*` pointer can produce unintended results.

Fix

One possible fix is to close the stream only at the end of operations. Another fix is to reopen the stream before using it again.

Example - Use of `FILE*` Pointer After Closing Stream

```
#include <stdio.h>

void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fclose(fp);
        fprintf(fp,"text");
    }
}
```

In this example, `fclose` closes the stream associated with `fp`. When you use `fprintf` on `fp` after `fclose`, the **Use of previously closed resource** defect appears.

Correction — Close Stream After All Operations

One possible correction is to reverse the order of the `fprintf` and `fclose` operations.

```
#include <stdio.h>

void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fprintf(fp,"text");
        fclose(fp);
    }
}
```

Check Information

Group: Rule 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FIO46-C

Introduced in R2019a

CERT C: Rule FIO47-C

Use valid format strings

Description

Rule Definition

Use valid format strings.

Polyspace Implementation

This checker checks for **Format string specifiers and arguments mismatch**.

Examples

Format string specifiers and arguments mismatch

Issue

Format string specifiers and arguments mismatch occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
  const char *message = "License not available";
  int err_number = ;-4
  printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Printing a Float

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", fst);
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

Check Information

Group: Rule 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FIO47-C

Introduced in R2019a

CERT C: Rule ENV30-C

Do not modify the object referenced by the return value of certain functions

Description

Rule Definition

Do not modify the object referenced by the return value of certain functions.

Polyspace Implementation

This checker checks for **Modification of internal buffer returned from nonreentrant standard function**.

Examples

Modification of internal buffer returned from nonreentrant standard function

Issue

Modification of internal buffer returned from nonreentrant standard function occurs when the following happens:

- A nonreentrant standard function returns a pointer.
- You attempt to write to the memory location that the pointer points to.

Nonreentrant standard functions that return a non `const`-qualified pointer to an internal buffer include `getenv`, `getlogin`, `crypt`, `setlocale`, `localeconv`, `strerror` and others.

Risk

Modifying the internal buffer that a nonreentrant standard function returns can cause the following issues:

- It is possible that the modification does not succeed or alters other internal data.

  For instance, `getenv` returns a pointer to an environment variable value. If you modify this value, you alter the environment of the process and corrupt other internal data.
- Even if the modification succeeds, it is possible that a subsequent call to the same standard function does not return your modified value.

  For instance, you modify the environment variable value that `getenv` returns. If another process, thread, or signal handler calls `setenv`, the modified value is overwritten. Therefore, a subsequent call to `getenv` does not return your modified value.

Fix

Avoid modifying the internal buffer using the pointer returned from the function.

Example - Modification of `getenv` Return Value

```
#include <stdlib.h>
#include <string.h>
```

```
void printstr(const char*);

void func() {
    char* env = getenv("LANGUAGE");
    if (env != NULL) {
        strncpy(env, "C", 1);
        printstr(env);
    }
}
```

In this example, the first argument of `strncpy` is the return value from a nonreentrant standard function `getenv`. The behavior can be undefined because `strncpy` modifies this argument.

Correction - Copy Return Value of `getenv` and Modify Copy

One possible solution is to copy the return value of `getenv` and pass the copy to the `strncpy` function.

```
#include <stdlib.h>
#include <string.h>
enum {
    SIZE20 = 20
};

void printstr(const char*);

void func() {
    char* env = getenv("LANGUAGE");
    if (env != NULL) {
        char env_cp[SIZE20];
        strncpy(env_cp, env, SIZE20);
        strncpy(env_cp, "C", 1);
        printstr(env_cp);
    }
}
```

Check Information

Group: Rule 10. Environment (ENV)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

ENV30-C

Introduced in R2019a

CERT C: Rule ENV31-C

Do not rely on an environment pointer following an operation that may invalidate it

Description

Rule Definition

Do not rely on an environment pointer following an operation that may invalidate it.

Polyspace Implementation

This checker checks for **Environment pointer invalidated by previous operation**.

Examples

Environment pointer invalidated by previous operation

Issue

Environment pointer invalidated by previous operation occurs when you use the third argument of *main()* in a hosted environment to access the environment after an operation modifies the environment. In a hosted environment, many C implementations support the nonstandard syntax:

```
main (int argc, char *argv[], char *envp[])
```

A call to a `setenv` or `putenv` family function modifies the environment pointed to by `*envp`.

Risk

When you modify the environment through a call to a `setenv` or `putenv` family function, the environment memory can potentially be reallocated. The hosted environment pointer is not updated and might point to an incorrect location. A call to this pointer can return unexpected results or cause an abnormal program termination.

Fix

Do not use the hosted environment pointer. Instead, use global external variable `environ` in Linux, `_environ` or `_wenviron` in Windows, or their equivalent. When you modify the environment, these variables are updated.

Example - Access Environment Through Pointer envp

```
#include <stdio.h>
#include <stdlib.h>

extern int check_arguments(int argc, char **argv, char **envp);
extern void use_envp(char **envp);

/* envp is from main function */
int func(char **envp)
{
    /* Call to setenv may cause environment
     *memory to be reallocated
     */
```

```
    if (setenv(("MY_NEW_VAR"),("new_value"),1) != 0)
    {
        /* Handle error */
        return -1;
    }
    /* envp not updated after call to setenv, and may
     *point to incorrect location.
     **/
    if (envp != ((void *)0)) {
        use_envp(envp);
/* No defect on second access to
*envp because defect already raised */
    }
    return 0;
}

void  main(int argc, char **argv, char **envp)
{
    if (check_arguments(argc, argv, envp))
    {
        (void)func(envp);
    }
}
```

In this example, `envp` is accessed inside `func()` after a call to `setenv` that can reallocate the environment memory. `envp` can point to an incorrect location because it is not updated after `setenv` modifies the environment. No defect is raised when `use_envp()` is called because the defect is already raised on the previous line of code.

Correction — Use Global External Variable `environ`

One possible correction is to access the environment by using a variable that is always updated after a call to `setenv`. For instance, in the following code, the pointer `envp` is still available from `main()`, but the environment is accessed in `func()` through the global external variable `environ`.

```
#include <stdio.h>
#include <stdlib.h>
extern char **environ;

extern int check_arguments(int argc, char **argv, char **envp);
extern void use_envp(char **envp);

int func(void)
{
    if (setenv(("MY_NEW_VAR"), ("new_value"),1) != 0) {
        /* Handle error */
        return -1;
    }
  /* Use global external variable environ
   *which is always updated after a call to setenv */

    if (environ != NULL) {
        use_envp(environ);
    }
    return 0;
}

void  main(int argc, char **argv, char **envp)
```

```
{
    if (check_arguments(argc, argv, envp))
    {
        (void)func();
    }
}
```

Check Information

Group: Rule 10. Environment (ENV)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

ENV31-C

Introduced in R2019a

CERT C: Rule ENV32-C

All exit handlers must return normally

Description

Rule Definition

All exit handlers must return normally.

Polyspace Implementation

This checker checks for **Abnormal termination of exit handler**.

Examples

Abnormal termination of exit handler

Issue

Abnormal termination of exit handler looks for registered exit handlers. Exit handlers are registered with specific functions such as `atexit`, (WinAPI) `_onexit`, or `at_quick_exit()`. If the exit handler calls a function that interrupts the program's expected termination sequence, Polyspace raises a defect. Some functions that can cause abnormal exits are `exit`, `abort`, `longjmp`, or (WinAPI) `_onexit`.

Risk

If your exit handler terminates your program, you can have undefined behavior. Abnormal program termination means other exit handlers are not invoked. These additional exit handlers may do additional clean up or other required termination steps.

Fix

In inside exit handlers, remove calls to functions that prevent the exit handler from terminating normally.

Example - Exit Handler With Call to `exit`

```
#include <stdlib.h>

volatile int some_condition = 1;
void demo_exit1(void)
{
    /* ... Cleanup code ... */
    return;
}
void exitabnormalhandler(void)
{
    if (some_condition)
    {
        /* Clean up */
        exit(0);
    }
    return;
```

```
}

int demo_install_exitabnormalhandler(void)
{

    if (atexit(demo_exit1) != 0) /* demo_exit1() performs additional cleanup */
    {
        /* Handle error */
    }
    if (atexit(exitabnormalhandler) != 0)
    {
        /* Handle error */
    }
    /* ... Program code ... */
    return 0;
}
```

In this example, `demo_install_exitabnormalhandler` registers two exit handlers, `demo_exit1` and `exitabnormalhandler`. Exit handlers are invoked in the reverse order of which they are registered. When the program ends, `exitabnormalhandler` runs, then `demo_exit1`. However, `exitabnormalhandler` calls `exit` interrupting the program exit process. Having this `exit` inside an exit handler causes undefined behavior because the program is not finished cleaning up safely.

Correction — Remove `exit` from Exit Handler

One possible correction is to let your exit handlers terminate normally. For this example, `exit` is removed from `exitabnormalhandler`, allowing the exit termination process to complete as expected.

```
#include <stdlib.h>

volatile int some_condition = 1;
void demo_exit1(void)
{
    /* ... Cleanup code ... */
    return;
}
void exitabnormalhandler(void)
{
    if (some_condition)
    {
        /* Clean up */
        /* Return normally */
    }
    return;
}

int demo_install_exitabnormalhandler(void)
{

    if (atexit(demo_exit1) != 0) /* demo_exit1() continues clean up */
    {
        /* Handle error */
    }
    if (atexit(exitabnormalhandler) != 0)
    {
        /* Handle error */
    }
```

```
    /* ... Program code ... */
    return 0;
}
```

Check Information

Group: Rule 10. Environment (ENV)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

ENV32-C

Introduced in R2019a

CERT C: Rule ENV33-C

Do not call system()

Description

Rule Definition

Do not call system().

Polyspace Implementation

This checker checks for **Unsafe call to a system function**.

Examples

Unsafe call to a system function

Issue

Unsafe call to a system function occurs when you use a function that invokes an implementation-defined command processor. These functions include:

- The C standard `system()` function.
- The POSIX `popen()` function.
- The Windows `_popen()` and `_wpopen()` functions.

Risk

If the argument of a function that invokes a command processor is not sanitized, it can cause exploitable vulnerabilities. An attacker can execute arbitrary commands or read and modify data anywhere on the system.

Fix

Do not use a `system`-family function to invoke a command processor. Instead, use safer functions such as POSIX `execve()` and WinAPI `CreateProcess()`.

Example - `system()` Called

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};

void func(char *arg)
{
    char buf[SIZE512];
    int retval=sprintf(buf, "/usr/bin/any_cmd %s", arg);
```

```
    if (retval<=0 || retval>SIZE512){
        /* Handle error */
        abort();
    }
    /* Use of system() to pass any_cmd with
    unsanitized argument to command processor */

    if (system(buf) == -1) {
    /* Handle error */
  }
}
```

In this example, `system()` passes its argument to the host environment for the command processor to execute. This code is vulnerable to an attack by command-injection.

Correction — Sanitize Argument and Use `execve()`

In the following code, the argument of `any_cmd` is sanitized, and then passed to `execve()` for execution. `exec`-family functions are not vulnerable to command-injection attacks.

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};


void func(char *arg)
{
  char *const args[SIZE3] = {"any_cmd", arg, NULL};
  char  *const env[] = {NULL};

  /* Sanitize argument */

  /* Use execve() to execute any_cmd. */

  if (execve("/usr/bin/time", args, env) == -1) {
    /* Handle error */
  }
}
```

Check Information

Group: Rule 10. Environment (ENV)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

ENV33-C

Introduced in R2019a

CERT C: Rule ENV34-C

Do not store pointers returned by certain functions

Description

Rule Definition

Do not store pointers returned by certain functions.

Polyspace Implementation

This checker checks for **Misuse of return value from nonreentrant standard function**.

Examples

Misuse of return value from nonreentrant standard function

Issue

Misuse of return value from nonreentrant standard function occurs when these events happen in this sequence:

1. You point to the buffer returned from a nonreentrant standard function such as `getenv` or `setlocale`.

   ```
   user = getenv("USER");
   ```

2. You call that nonreentrant standard function again.

   ```
   user2 = getenv("USER2");
   ```

3. You use or dereference the pointer from the first step expecting the buffer to remain unmodified since that step. In the meantime, the call in the second step has modified the buffer.

   For instance:

   ```
   var=*user;
   ```

In some cases, the defect might appear even if you do not call the `getenv` function a second time but simply return the pointer. For instance:

```
char* func() {
     user=getenv("USER");
     .
     .
     return user;
}
```

For information on which functions are covered by this defect, see documentation on nonreentrant standard functions.

Risk

The C Standard allows nonreentrant functions such as `getenv` to return a pointer to a *static* buffer. Because the buffer is static, a second call to `getenv` modifies the buffer. If you continue to use the

pointer returned from the first call past the second call, you can see unexpected results. The buffer that it points to no longer has values from the first call.

The defect appears even if you do not call `getenv` a second time but simply return the pointer. The reason is that someone calling your function might use the returned pointer *after* a second call to `getenv`. By returning the pointer from your call to `getenv`, you make your function unsafe to use.

The same rationale is true for other nonreentrant functions covered by this defect.

Fix

After the first call to `getenv`, make a copy of the buffer that the returned pointer points to. After the second call to `getenv`, use this copy. Even if the second call modifies the buffer, your copy is untouched.

Example - Return from getenv Used After Second Call to getenv

```
#include <stdlib.h>
#include <string.h>

int func()
{
    int result = 0;

    char *home = getenv("HOME");   /* First call */
    if (home != NULL) {
        char *user = NULL;
        char *user_name_from_home = strrchr(home, '/');

        if (user_name_from_home != NULL) {
            user = getenv("USER");   /* Second call */
            if ((user != NULL) &&
                (strcmp(user, user_name_from_home) == 0))
            {
                result = 1;
            }
        }
    }
    return result;
}
```

In this example, the pointer `user_name_from_home` is derived from the pointer `home`. `home` points to the buffer returned from the first call to `getenv`. Therefore, `user_name_from_home` points to a location in the same buffer.

After the second call to `getenv`, the buffer is modified. If you continue to use `user_name_from_home`, you can get unexpected results.

Correction — Make Copy of Buffer Before Second Call

If you want to access the buffer from the first call to `getenv` past the second call, make a copy of the buffer after the first call. One possible correction is to use the `strdup` function to make the copy.

```
#include <stdlib.h>
#include <string.h>

int func()
{
```

```
    int result = 0;

    char *home = getenv("HOME");
    if (home != NULL) {
        char *user = NULL;
        char *user_name_from_home = strrchr(home, '/');
        if (user_name_from_home != NULL) {
            /* Make copy before second call */
            char *saved_user_name_from_home = strdup(user_name_from_home);
            if (saved_user_name_from_home != NULL) {
                user = getenv("USER");
                if ((user != NULL) &&
                    (strcmp(user, saved_user_name_from_home) == 0))
                {
                    result = 1;
                }
                free(saved_user_name_from_home);
            }
        }
    }
    return result;
}
```

Check Information

Group: Rule 10. Environment (ENV)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

ENV34-C

Introduced in R2019a

CERT C: Rule SIG30-C

Call only asynchronous-safe functions within signal handlers

Description

Rule Definition

Call only asynchronous-safe functions within signal handlers.

Polyspace Implementation

This checker checks for these issues:

- **Function called from signal handler not asynchronous-safe**.
- **Function called from signal handler not asynchronous-safe (strict)**.

Examples

Function called from signal handler not asynchronous-safe

Issue

Function called from signal handler not asynchronous-safe occurs when a signal handler calls a function that is not asynchronous-safe according to the POSIX standard. An asynchronous-safe function can be interrupted at any point in its execution, then called again without causing an inconsistent state. It can also correctly handle global data that might be in an inconsistent state.

If a signal handler calls another function that calls an asynchronous-unsafe function, the defect appears on the function call in the signal handler. The defect traceback shows the full path from the signal handler to the asynchronous-unsafe function.

Risk

When a signal handler is invoked, the execution of the program is interrupted. After the handler is finished, program execution resumes at the point of interruption. If a function is executing at the time of the interruption, calling it from within the signal handler is undefined behavior, unless it is asynchronous-safe.

Fix

The POSIX standard defines these functions as asynchronous-safe. You can call these functions from a signal handler.

`_exit()`	`getpgrp()`	`setsockopt()`
`_Exit()`	`getpid()`	`setuid()`
`abort()`	`getppid()`	`shutdown()`
`accept()`	`getsockname()`	`sigaction()`
`access()`	`getsockopt()`	`sigaddset()`
`aio_error()`	`getuid()`	`sigdelset()`

aio_return()	kill()	sigemptyset()
aio_suspend()	link()	sigfillset()
alarm()	linkat()	sigismember()
bind()	listen()	signal()
cfgetispeed()	lseek()	sigpause()
cfgetospeed()	lstat()	sigpending()
cfsetispeed()	mkdir()	sigprocmask()
cfsetospeed()	mkdirat()	sigqueue()
chdir()	mkfifo()	sigset()
chmod()	mkfifoat()	sigsuspend()
chown()	mknod()	sleep()
clock_gettime()	mknodat()	sockatmark()
close()	open()	socket()
connect()	openat()	socketpair()
creat()	pathconf()	stat()
dup()	pause()	symlink()
dup2()	pipe()	symlinkat()
execl()	poll()	sysconf()
execle()	posix_trace_event()	tcdrain()
execv()	pselect()	tcflow()
execve()	pthread_kill()	tcflush()
faccessat()	pthread_self()	tcgetattr()
fchdir()	pthread_sigmask()	tcgetpgrp()
fchmod()	quick_exit()	tcsendbreak()
fchmodat()	raise()	tcsetattr()
fchown()	read()	tcsetpgrp()
fchownat()	readlink()	time()
fcntl()	readlinkat()	timer_getoverrun()
fdatasync()	recv()	timer_gettime()
fexecve()	recvfrom()	timer_settime()
fork()	recvmsg()	times()
fpathconf()	rename()	umask()
fstat()	renameat()	uname()
fstatat()	rmdir()	unlink()
fsync()	select()	unlinkat()
ftruncate()	sem_post()	utime()
futimens()	send()	utimensat()

getegid()	sendmsg()	utimes()
geteuid()	sendto()	wait()
getgid()	setgid()	waitpid()
getgroups()	setpgid()	write()
getpeername()	setsid()	

Functions not in the previous table are not asynchronous-safe, and should not be called from a signal hander.

Example - Call to `printf()` Inside Signal Handler

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

#define SIZE20 20

extern volatile sig_atomic_t e_flag;

void display_info(const char *info)
{
    if (info)
    {
        (void)fputs(info, stderr);
    }
}

void sig_handler(int signum)
{
    /* Call function printf() that is not
    asynchronous-safe */
    printf("signal %d received.", signum);
    e_flag = 1;
}

int main(void)
{
    e_flag = 0;
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info = (char *)calloc(SIZE20, sizeof(char));
    if (info == NULL)
    {
        /* Handle Error */
    }
    while (!e_flag)
    {
        /* Main loop program code */
        display_info(info);
        /* More program code */
```

```
    }
    free(info);
    info = NULL;
    return 0;
}
```

In this example, `sig_handler` calls `printf()` when catching a signal. If the handler catches another signal while `printf()` is executing, the behavior of the program is undefined.

Correction — Set Flag Only in Signal Handler

Use your signal handler to set only the value of a flag. `e_flag` is of type volatile `sig_atomic_t`. `sig_handler` can safely access it asynchronously.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

#define SIZE20 20

extern volatile sig_atomic_t e_flag;

void display_info(const char *info)
{
    if (info)
    {
        (void)fputs(info, stderr);
    }
}

void sig_handler1(int signum)
{
    int s0 = signum;
    e_flag = 1;
}

int func(void)
{
    e_flag = 0;
    if (signal(SIGINT, sig_handler1) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info = (char *)calloc(SIZE20, 1);
    if (info == NULL)
    {
        /* Handle error */
    }
    while (!e_flag)
    {
        /* Main loop program code */
        display_info(info);
```

```
        /* More program code */
    }
    free(info);
    info = NULL;
    return 0;
}
```

Function called from signal handler not asynchronous-safe (strict)

Issue

Function called from signal handler not asynchronous-safe (strict) occurs when a signal handler calls a function that is not asynchronous-safe according to the C standard. An asynchronous-safe function can be interrupted at any point in its execution, then called again without causing an inconsistent state. It can also correctly handle global data that might be in an inconsistent state.

When you select the checker **Function called from signal handler not asynchronous-safe**, the checker detects calls to functions that are not asynchronous-safe according to the POSIX standard. **Function called from signal handler not asynchronous-safe (strict)** does not raise a defect for these cases. **Function called from signal handler not asynchronous-safe (strict)** raises a defect for functions that are asynchronous-safe according to the POSIX standard but not according to the C standard.

If a signal handler calls another function that calls an asynchronous-unsafe function, the defect appears on the function call in the signal handler. The defect traceback shows the full path from the signal handler to the asynchronous-unsafe function.

Risk

When a signal handler is invoked, the execution of the program is interrupted. After the handler is finished, program execution resumes at the point of interruption. If a function is executing at the time of the interruption, calling it from within the signal handler is undefined behavior, unless it is asynchronous-safe.

Fix

The C standard defines the following functions as asynchronous-safe. You can call these functions from a signal handler:

- `abort()`
- `_Exit()`
- `quick_exit()`
- `signal()`

Example - Call to `raise()` Inside Signal Handler

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

void SIG_ERR_handler(int signum)
{
```

```
    int s0 = signum;
    /* SIGTERM specific handling */
}

void sig_handler(int signum)
{
    int s0 = signum;
    /* Call raise() */
    if (raise(SIGTERM) != 0) {
        /* Handle error */
    }
}

int finc(void)
{
    if (signal(SIGTERM, SIG_ERR_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return 0;
}
```

In this example, `sig_handler` calls `raise()` when catching a signal. If the handler catches another signal while `raise()` is executing, the behavior of the program is undefined.

Correction — Remove Call to `raise()` in Signal Handler

According to the C standard, the only functions that you can safely call from a signal handler are `abort()`, `_Exit()`, `quick_exit()`, and `signal()`.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

void SIG_ERR_handler(int signum)
{
    int s0 = signum;
    /* SIGTERM specific handling */
}
void sig_handler(int signum)
{
    int s0 = signum;
```

```
}

int func(void)
{
    if (signal(SIGTERM, SIG_ERR_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return 0;
}
```

Check Information

Group: Rule 11. Signals (SIG)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

SIG30-C

Introduced in R2019a

CERT C: Rule SIG31-C

Do not access shared objects in signal handlers

Description

Rule Definition

Do not access shared objects in signal handlers.

Polyspace Implementation

This checker checks for **Shared data access within signal handler**.

Examples

Shared data access within signal handler

Issue

Shared data access within signal handler occurs when you access or modify a shared object inside a signal handler.

Risk

When you define a signal handler function to access or modify a shared object, the handler accesses or modifies the shared object when it receives a signal. If another function is already accessing the shared object, that function causes a race condition and can leave the data in an inconsistent state.

Fix

To access or modify shared objects inside a signal handler, check that the objects are lock-free atomic, or, if they are integers, declare them as `volatile sig_atomic_t`.

Example - `int` Variable Access in Signal Handler

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* declare global variable. */
int e_flag;

void sig_handler(int signum)
{
    /* Signal handler accesses variable that is not
    of type volatile sig_atomic_t. */
    e_flag = signum;
}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
```

```
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

In this example, `sig_handler` accesses `e_flag`, a variable of type `int`. A concurrent access by another function can leave `e_flag` in an inconsistent state.

Correction — Declare Variable of Type `volatile sig_atomic_t`

Before you access a shared variable from a signal handler, declare the variable with type `volatile sig_atomic_t` instead of `int`. You can safely access variables of this type asynchronously.

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* Declare variable of type volatile sig_atomic_t. */
volatile sig_atomic_t e_flag;
void sig_handler(int signum)
{
    /* Use variable of proper type inside signal handler. */
    e_flag = signum;

}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

Check Information

Group: Rule 11. Signals (SIG)

See Also

`Check SEI CERT-C (-cert-c)`

Topics
“Check for Coding Standard Violations”

External Websites
SIG31-C

Introduced in R2019a

CERT C: Rule SIG34-C

Do not call signal() from within interruptible signal handlers

Description

Rule Definition

Do not call signal() from within interruptible signal handlers.

Polyspace Implementation

This checker checks for **Signal call from within signal handler**.

Examples

Signal call from within signal handler

Issue

Signal call from within signal handler occurs when you call `signal()` from a nonpersistent signal handler on a Windows platform.

Risk

A nonpersistent signal handler is reset after catching a signal. The handler does not catch subsequent signals unless the handler is reestablished by calling `signal()`. A nonpersistent signal handler on a Windows platform is reset to `SIG_DFL`. If another signal interrupts the execution of the handler, that signal can cause a race condition between `SIG_DFL` and the existing signal handler. A call to `signal()` can also result in an infinite loop inside the handler.

Fix

Do not call `signal()` from a signal handler on Windows platforms.

Example - `signal()` Called from Signal Handler

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;

void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;

    /* Call signal() to reestablish sig_handler
    upon receiving SIG_ERR. */

    if (signal(s0, sig_handler) == SIG_ERR)
```

```
    {
        /* Handle error */
    }
}

void func(void)
{
        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */

        }
  /* more code */
}
```

In this example, the definition of `sig_handler()` includes a call to `signal()` when the handler catches `SIG_ERR`. On Windows platforms, signal handlers are nonpersistent. This code can result in a race condition.

Correction — Do Not Call `signal()` from Signal Handler

If your code requires the use of a persistent signal handler on a Windows platform, use a persistent signal handler after performing a thorough risk analysis.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;


void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;
    /* No call to signal() */
}

int main(void)
{

        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */

        }
}
```

Check Information

Group: Rule 11. Signals (SIG)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

SIG34-C

Introduced in R2019a

CERT C: Rule SIG35-C

Do not return from a computational exception signal handler

Description

Rule Definition

Do not return from a computational exception signal handler.

Polyspace Implementation

This checker checks for **Return from computational exception signal handler**.

Examples

Return from computational exception signal handler

Issue

Return from computational exception signal handler occurs when a signal handler returns after catching a computational exception signal `SIGFPE`, `SIGILL`, or `SIGSEGV`.

Risk

A signal handler that returns normally from a computational exception is undefined behavior. Even if the handler attempts to fix the error that triggered the signal, the program can behave unexpectedly.

Fix

Check the validity of the values of your variables before the computation to avoid using a signal handler to catch exceptions. If you cannot avoid a handler to catch computation exception signals, call `abort()`, `quick_exit()`, or `_Exit()` in the handler to stop the program.

Example - Signal Handler Return from Division by Zero

```
#include <errno.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>

static volatile sig_atomic_t denom;
/* Declare signal handler to catch division by zero
computation error. */
void sig_handler(int s)
{
    int s0 = s;
    if (denom == 0)
    {
        denom = 1;
    }
    /* Normal return from computation exception
    signal */
    return;
}
```

```
long func(int v)
{
    denom = (sig_atomic_t)v;

        if (signal(SIGFPE, sig_handler) == SIG_ERR)
        {
            /* Handle error */
        }

    long result = 100 / (long)denom;
    return result;
}
```

In this example, `sig_handler` is declared to handle a division by zero computation error. The handler changes the value of `denom` if it is zero and returns, which is undefined behavior.

Correction — Call `abort()` to Terminate Program

After catching a computational exception, call `abort()` from `sig_handler` to exit the program without further error.

```
#include <errno.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>

static volatile sig_atomic_t denom;
/* Declare signal handler to catch division by zero
computation error. */

void sig_handler(int s)
{
    int s0 = s;
    /* call to abort() to exit the program */
    abort();
}

long func(int v)
{
    denom = (sig_atomic_t)v;

        if (signal(SIGFPE, sig_handler) == SIG_ERR)
        {
            /* Handle error */
        }

    long result = 100 / (long)denom;
    return result;
}
```

Check Information

Group: Rule 11. Signals (SIG)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

SIG35-C

Introduced in R2019a

CERT C: Rule ERR30-C

Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure

Description

Rule Definition

Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure.

Polyspace Implementation

This checker checks for these issues:

- **Misuse of errno**.
- **Errno not reset**.

Examples

Misuse of errno

Issue

Misuse of errno occurs when you check `errno` for error conditions in situations where checking `errno` does not guarantee the absence of errors. In some cases, checking `errno` can lead to false positives.

For instance, you check `errno` following calls to the functions:

- `fopen`: If you follow the ISO Standard, the function might not set `errno` on errors.
- `atof`: If you follow the ISO Standard, the function does not set `errno`.
- `signal`: The `errno` value indicates an error only if the function returns the `SIG_ERR` error indicator.

Risk

The ISO C Standard does not enforce that these functions set `errno` on errors. Whether the functions set `errno` or not is implementation-dependent.

To detect errors, if you check `errno` alone, the validity of this check also becomes implementation-dependent.

In some cases, the `errno` value indicates an error only if the function returns a specific error indicator. If you check `errno` before checking the function return value, you can see false positives.

Fix

For information on how to detect errors, see the documentation for that specific function.

Typically, the functions return an out-of-band error indicator to indicate errors. For instance:

- `fopen` returns a null pointer if an error occurs.
- `signal` returns the `SIG_ERR` error indicator and sets `errno` to a positive value. Check `errno` only after you have checked the function return value.

Example - Incorrectly Checking for `errno` After `fopen` Call

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define fatal_error() abort()

const char *temp_filename = "/tmp/demo.txt";

FILE *func()
{
    FILE *fileptr;
    errno = 0;
    fileptr = fopen(temp_filename, "w+b");
    if (errno != 0) {
        if (fileptr != NULL) {
            (void)fclose(fileptr);
        }
        /* Handle error */
        fatal_error();
    }
    return fileptr;
}
```

In this example, `errno` is the first variable that is checked after a call to `fopen`. You might expect that `fopen` changes `errno` to a nonzero value if an error occurs. If you run this code with an implementation of `fopen` that does not set `errno` on errors, you might miss an error condition. In this situation, `fopen` can return a null pointer that escapes detection.

Correction — Check Return Value of `fopen` After Call

One possible correction is to only check the return value of `fopen` for a null pointer.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define fatal_error() abort()

const char *temp_filename = "/tmp/demo.txt";

FILE *func()
{
    FILE *fileptr;
    fileptr = fopen(temp_filename, "w+b");
    if (fileptr == NULL) {
        fatal_error();
    }
    return fileptr;
}
```

Errno not reset

Issue

Errno not reset occurs when you do not reset `errno` before calling a function that sets `errno` to indicate error conditions. However, you check `errno` for those error conditions after the function call.

Risk

The `errno` is not clean and can contain values from a previous call. Checking `errno` for errors can give the false impression that an error occurred.

`errno` is set to zero at program startup but subsequently, `errno` is not reset by a C standard library function. You must explicitly set `errno` to zero when required.

Fix

Before calling a function that sets `errno` to indicate error conditions, reset `errno` to zero explicitly.

Example - `errno` Not Reset Before Call to `strtod`

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>

#define fatal_error() abort()

double func(const char *s1, const char *s2)
{
    double f1;
    f1 = strtod (s1, NULL);
    if (0 == errno) {
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
                  {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

In this example, `errno` is not reset to 0 before the first call to `strtod`. Checking `errno` for 0 later can lead to a false positive.

Correction — Reset `errno` Before Call

One possible correction is to reset `errno` to 0 before calling `strtod`.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>
```

```
#define fatal_error() abort()

double func(const char *s1, const char *s2)
{
    double f1;
    errno = 0;
    f1 = strtod (s1, NULL);
    if (0 == errno) {
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
              {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

Check Information

Group: Rule 12. Error Handling (ERR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

ERR30-C

Introduced in R2019a

CERT C: Rule ERR32-C

Do not rely on indeterminate values of errno

Description

Rule Definition

Do not rely on indeterminate values of errno.

Polyspace Implementation

This checker checks for **Misuse of errno in a signal handler**.

Examples

Misuse of errno in a signal handler

Issue

Misuse of errno in a signal handler occurs when you call one of these functions in a signal handler:

- `signal`: You call the `signal` function in a signal handler and then read the value of `errno`.

  For instance, the signal handler function `handler` calls `signal` and then calls `perror`, which reads `errno`.

  ```
  void handler(int signum) {
    pfv old_handler = signal(signum, SIG_DFL);
    if (old_handler == SIG_ERR) {
      perror("SIGINT handler");
    }
  }
  ```

- `errno`-setting POSIX function: You call an `errno`-setting POSIX function in a signal handler but do not restore `errno` when returning from the signal handler.

  For instance, the signal handler function `handler` calls `waitpid`, which changes `errno`, but does not restore `errno` before returning.

  ```
  void handler(int signum) {
    int rc = waitpid(-1, NULL, WNOHANG);
    if (ECHILD != errno) {
    }
  }
  ```

Risk

In each case that the checker flags, you risk relying on an indeterminate value of `errno`.

- `signal`: If the call to `signal` in a signal handler fails, the value of `errno` is indeterminate (see C11 Standard, Sec. 7.14.1.1). If you rely on a specific value of `errno`, you can see unexpected results.

- `errno`-setting POSIX function: An `errno`-setting function sets `errno` on failure. If you read `errno` after a signal handler is called and the signal handler itself calls an `errno`-setting function, you can see unexpected results.

Fix

Avoid situations where you risk relying on an indeterminate value of `errno`.

- `signal`: After calling the `signal` function in a signal handler, do not read `errno` or use a function that reads `errno`.
- `errno`-setting POSIX function: Before calling an `errno`-setting function in a signal handler, save `errno` to a temporary variable. Restore `errno` from this variable before returning from the signal handler.

Example - Reading `errno` After `signal` Call in Signal Handler

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

#define fatal_error() abort()

void handler(int signum) {
    if (signal(signum, SIG_DFL) == SIG_ERR) {
        perror("SIGINT handler");
    }
}

int func(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
        fatal_error();
    }
    /* Program code */
    if (raise(SIGINT) != 0) {
        /* Handle error */
        fatal_error();
    }
    return 0;
}
```

In this example, the function `handler` is called to handle the `SIGINT` signal. In the body of `handler`, the `signal` function is called. Following this call, the value of `errno` is indeterminate. The checker raises a defect when the `perror` function is called because `perror` relies on the value of `errno`.

Correction — Avoid Reading `errno` After `signal` Call

One possible correction is to not read `errno` after calling the `signal` function in a signal handler. The corrected code here calls the `abort` function via the `fatal_error` macro instead of the `perror` function.

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

#define fatal_error() abort()
```

```
void handler(int signum) {
    if (signal(signum, SIG_DFL) == SIG_ERR) {
        fatal_error();
    }
}

int func(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
        fatal_error();
    }
    /* Program code */
    if (raise(SIGINT) != 0) {
        /* Handle error */
        fatal_error();
    }
    return 0;
}
```

Check Information

Group: Rule 12. Error Handling (ERR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

ERR32-C

Introduced in R2019a

CERT C: Rule ERR33-C

Detect and handle standard library errors

Description

Rule Definition

Detect and handle standard library errors.

Polyspace Implementation

This checker checks for these issues:

- **Errno not checked**.
- **Returned value of a sensitive function not checked**.
- **Unprotected dynamic memory allocation**.

Examples

Errno not checked

Issue

Errno not checked occurs when you call a function that sets `errno` to indicate error conditions, but do not check `errno` after the call. For these functions, checking `errno` is the only reliable way to determine if an error occurred.

Functions that set `errno` on errors include:

- `fgetwc`, `strtol`, and `wcstol`.

  For a comprehensive list of functions, see documentation about errno.
- POSIX `errno`-setting functions such as `encrypt` and `setkey`.

Risk

To see if the function call completed without errors, check `errno` for error values.

The return values of these `errno`-setting functions do not indicate errors. The return value can be one of the following:

- `void`
- Even if an error occurs, the return value can be the same as the value from a successful call. Such return values are called in-band error indicators.

You can determine if an error occurred only by checking `errno`.

For instance, `strtol` converts a string to a long integer and returns the integer. If the result of conversion overflows, the function returns `LONG_MAX` and sets `errno` to `ERANGE`. However, the function can also return `LONG_MAX` from a successful conversion. Only by checking `errno` can you distinguish between an error and a successful conversion.

Fix

Before calling the function, set `errno` to zero.

After the function call, to see if an error occurred, compare `errno` to zero. Alternatively, compare `errno` to known error indicator values. For instance, `strtol` sets `errno` to `ERANGE` to indicate errors.

The error message in the Polyspace result shows the error indicator value that you can compare to.

Example - `errno` Not Checked After Call to `strtol`

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    long val = strtol(str, &endptr, base);
    printf("Return value of strtol() = %ld\n", val);
}
```

You are using the return value of `strtol` without checking `errno`.

Correction — Check `errno` After Call

Before calling `strtol`, set `errno` to zero . After a call to `strtol`, check the return value for `LONG_MIN` or `LONG_MAX` and `errno` for `ERANGE`.

```
#include<stdlib.h>
#include<stdio.h>
#include<errno.h>
#include<limits.h>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    errno = 0;
    long val = strtol(str, &endptr, base);
    if((val == LONG_MIN || val == LONG_MAX) && errno == ERANGE) {
         printf("strtol error");
         exit(EXIT_FAILURE);
    }
    printf("Return value of strtol() = %ld\n", val);
}
```

Returned value of a sensitive function not checked

Issue

Returned value of a sensitive function not checked occurs when you call sensitive standard functions, but you:

- Ignore the return value.
- Use an output or a return value without testing the validity of the return value.

For this defect, two type of functions are considered: ***sensitive*** and ***critical sensitive***.

A ***sensitive*** function is a standard function that can encounter:

- Exhausted system resources (for example, when allocating resources)
- Changed privileges or permissions
- Tainted sources when reading, writing, or converting data from external sources
- Unsupported features despite an existing API

A ***critical sensitive*** function is a sensitive function that performs one of these critical or vulnerable tasks:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

Risk

If you do not check the return value of functions that perform sensitive or critical sensitive tasks, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

Fix

Before continuing with the program, test the return value of *critical sensitive* functions.

For *sensitive functions*, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

Example - Sensitive Function Return Ignored

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    pthread_attr_init(&attr);
}
```

This example shows a call to the sensitive function `pthread_attr_init`. The return value of `pthread_attr_init` is ignored, causing a defect.

Correction — Cast Function to (void)

One possible correction is to cast the function to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of the sensitive function.

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    (void)pthread_attr_init(&attr);
}
```

Correction — Test Return Value

One possible correction is to test the return value of `pthread_attr_init` to check for errors.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

void initialize() {
    pthread_attr_t attr;
    int result;

    result = pthread_attr_init(&attr);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

Example - Critical Function Return Ignored

```
#include <pthread.h>
extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0));
    pthread_join(thread_id,  &res);
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), Polyspace does not ignore this *Return value of a sensitive function not checked* defect. The other critical function, `pthread_join`, returns value that is ignored implicitly. `pthread_join` uses the return value of `pthread_create`, which was not checked.

Correction — Test the Return Value of Critical Functions

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

Unprotected dynamic memory allocation

Issue

Unprotected dynamic memory allocation occurs when you do not check after dynamic memory allocation whether the memory allocation succeeded.

Risk

When memory is dynamically allocated using `malloc`, `calloc`, or `realloc`, it returns a value `NULL` if the requested memory is not available. If the code following the allocation accesses the memory block without checking for this `NULL` value, this access is not protected from failures.

Fix

Check the return value of `malloc`, `calloc`, or `realloc` for NULL before accessing the allocated memory location.

```
int *ptr = malloc(size * sizeof(int));

if(ptr) /* Check for NULL */
{
   /* Memory access through ptr */
}
```

Example - Unprotected dynamic memory allocation error

```
#include <stdlib.h>

void Assign_Value(void)
{
  int* p = (int*)calloc(5, sizeof(int));
```

```
  *p = 2;
  /* Defect: p is not checked for NULL value */

  free(p);
}
```

If the memory allocation fails, the function `calloc` returns `NULL` to `p`. Before accessing the memory through `p`, the code does not check whether `p` is `NULL`

Correction — Check for NULL Value

One possible correction is to check whether `p` has value `NULL` before dereference.

```
#include <stdlib.h>

void Assign_Value(void)
 {
   int* p = (int*)calloc(5, sizeof(int));

   /* Fix: Check if p is NULL */
   if(p!=NULL) *p = 2;

   free(p);
 }
```

Check Information

Group: Rule 12. Error Handling (ERR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

ERR33-C

Introduced in R2019a

CERT C: Rule ERR34-C

Detect errors when converting a string to a number

Description

Rule Definition

Detect errors when converting a string to a number.

Polyspace Implementation

This checker checks for **Unsafe conversion from string to numerical value**.

Examples

Unsafe conversion from string to numerical value

Issue

Unsafe conversion from string to numerical value detects conversions from strings to integer or floating-point values. If your conversion method does not include robust error handling, a defect is raised.

Risk

Converting a string to numerical value can cause data loss or misinterpretation. Without validation of the conversion or error handling, your program continues with invalid values.

Fix

- Add additional checks to validate the numerical value.
- Use a more robust string-to-numeric conversion function such as `strtol`, `strtoll`, `strtoul`, or `strtoull`.

Example - Conversion With `atoi`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int demo_check_string_not_empty(char *s)
{
    if (s != NULL)
        return strlen(s) > 0; /* check string null-terminated and not empty */
    else
        return 0;
}

int unsafestrtonumeric(char* argv1)
{
    int s = 0;
    if (demo_check_string_not_empty(argv1))
    {
```

```
        s = atoi(argv1);
    }
    return s;
}
```

In this example, `argv1` is converted to an integer with `atoi`. `atoi` does not provide errors for an invalid integer string. The conversion can fail unexpectedly.

Correction — Use `strtol` instead

One possible correction is to use `strtol` to validate the input string and the converted integer.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <errno.h>

static int demo_check_string_not_empty(char *s)
{
    if (s != NULL)
        return strlen(s) > 0; /* check string null-terminated and not empty */
    else
        return 0;
}

int unsafestrtonumeric(char *argv1)
{
    char *c_str = argv1;
    char *end;
    long sl;

    if (demo_check_string_not_empty(c_str))
    {
        errno = 0; /* set errno for error check */
        sl = strtol(c_str, &end, 10);
        if (end == c_str)
        {
            (void)fprintf(stderr, "%s: not a decimal number\n", c_str);
        }
        else if ('\0' != *end)
        {
            (void)fprintf(stderr, "%s: extra characters: %s\n", c_str, end);
        }
        else if ((LONG_MIN == sl || LONG_MAX == sl) && ERANGE == errno)
        {
            (void)fprintf(stderr, "%s out of range of type long\n", c_str);
        }
        else if (sl > INT_MAX)
        {
            (void)fprintf(stderr, "%ld greater than INT_MAX\n", sl);
        }
        else if (sl < INT_MIN)
        {
            (void)fprintf(stderr, "%ld less than INT_MIN\n", sl);
        }
        else
        {
```

```
            return (int)sl;
        }
    }
    return 0;
}
```

Check Information

Group: Rule 12. Error Handling (ERR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

ERR34-C

Introduced in R2019a

CERT C: Rule CON30-C

Clean up thread-specific storage

Description

Rule Definition

Clean up thread-specific storage.

Polyspace Implementation

This checker checks for **Thread-specific memory leak**.

Examples

Thread-specific memory leak

Issue

Thread-specific memory leak occurs when you do not free thread-specific dynamically allocated memory before the end of a thread.

To create thread-specific storage, you generally do these steps:

1. You create a key for thread-specific storage.
2. You create the threads.
3. In each thread, you allocate storage dynamically and then associate the key with this storage.

   After the association, you can read the stored data later using the key.
4. Before the end of the thread, you free the thread-specific memory using the key.

The checker flags execution paths in the thread where the last step is missing.

The checker works on these families of functions:

- `tss_get` and `tss_set` (C11)
- `pthread_getspecific` and `pthread_setspecific` (POSIX)

Risk

The data stored in the memory is available to other processes even after the threads end (memory leak). Besides security vulnerabilities, memory leaks can shrink the amount of available memory and reduce performance.

Fix

Free dynamically allocated memory before the end of a thread.

You can explicitly free dynamically allocated memory with functions such as `free`.

Alternatively, when you create a key, you can associate a destructor function with the key. The destructor function is called with the key value as argument at the end of a thread. In the body of the

destructor function, you can free any memory associated with the key. If you use this method, Bug Finder still flags a defect. Ignore this defect with appropriate comments. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Memory Not Freed at End of Thread

```
#include <threads.h>
#include <stdlib.h>

/* Global key to the thread-specific storage */
tss_t key;
enum { MAX_THREADS = 3 };


int add_data(void) {
  int *data = (int *)malloc(2 * sizeof(int));
  if (data == NULL) {
    return -1;  /* Report error  */
  }
  data[0] = 0;
  data[1] = 1;

  if (thrd_success != tss_set(key, (void *)data)) {
    /* Handle error */
  }
  return 0;
}

void print_data(void) {
  /* Get this thread's global data from key */
  int *data = tss_get(key);

  if (data != NULL) {
    /* Print data */
  }
}

int func(void *dummy) {
  if (add_data() != 0) {
    return -1;  /* Report error */
  }
  print_data();
  return 0;
}

int main(void) {
  thrd_t thread_id[MAX_THREADS];

  /* Create the key before creating the threads */
  if (thrd_success != tss_create(&key, NULL)) {
    /* Handle error */
  }

  /* Create threads that would store specific storage */
  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_create(&thread_id[i], func, NULL)) {
      /* Handle error */
    }
```

```
  }

  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_join(thread_id[i], NULL)) {
      /* Handle error */
    }
  }

  tss_delete(key);
  return 0;
}
```

In this example, the start function of each thread `func` calls two functions:

- `add_data`: This function allocates storage dynamically and associates the storage with a key using the `tss_set` function.
- `print_data`: This function reads the stored data using the `tss_get` function.

At the points where `func` returns, the dynamically allocated storage has not been freed.

Correction — Free Dynamically Allocated Memory Explicitly

One possible correction is to free dynamically allocated memory explicitly before leaving the start function of a thread. See the highlighted change in the corrected version.

In this corrected version, a defect still appears on the `return` statement in the error handling section of `func`. The defect cannot occur in practice because the error handling section is entered only if dynamic memory allocation fails. Ignore this remaining defect with appropriate comments. See "Address Polyspace Results Through Bug Fixes or Justifications".

```
#include <threads.h>
#include <stdlib.h>

/* Global key to the thread-specific storage */
tss_t key;
enum { MAX_THREADS = 3 };


int add_data(void) {
  int *data = (int *)malloc(2 * sizeof(int));
  if (data == NULL) {
    return -1;  /* Report error  */
  }
  data[0] = 0;
  data[1] = 1;

  if (thrd_success != tss_set(key, (void *)data)) {
    /* Handle error */
  }
  return 0;
}

void print_data(void) {
  /* Get this thread's global data from key */
  int *data = tss_get(key);

  if (data != NULL) {
```

```
    /* Print data */
  }
}

int func(void *dummy) {
  if (add_data() != 0) {
    return -1;  /* Report error */
  }
  print_data();
  free(tss_get(key));
  return 0;
}

int main(void) {
  thrd_t thread_id[MAX_THREADS];

  /* Create the key before creating the threads */
  if (thrd_success != tss_create(&key, NULL)) {
    /* Handle error */
  }

  /* Create threads that would store specific storage */
  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_create(&thread_id[i], func, NULL)) {
      /* Handle error */
    }
  }

  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_join(thread_id[i], NULL)) {
      /* Handle error */
    }
  }

  tss_delete(key);
  return 0;
}
```

Check Information

Group: Rule 14. Concurrency (CON)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

CON30-C

Introduced in R2019a

CERT C: Rule CON31-C

Do not destroy a mutex while it is locked

Description

Rule Definition

Do not destroy a mutex while it is locked.

Polyspace Implementation

This checker checks for **Destruction of locked mutex**.

Examples

Destruction of locked mutex

Issue

Destruction of locked mutex occurs when a task destroys a mutex after it is locked (and before it is unlocked). The locking and destruction can happen in the same task or different tasks.

Risk

A mutex is locked to protect shared variables from concurrent access. If a mutex is destroyed in the locked state, the protection does not apply.

Fix

To fix this defect, destroy the mutex only after you unlock it. It is a good design practice to:

- Initialize a mutex *before* creating the threads where you use the mutex.
- Destroy a mutex *after* joining the threads that you created.

On the **Result Details** pane, you see two events, the locking and destruction of the mutex, and the tasks that initiated the events. To navigate to the corresponding line in your source code, click the event.

Example - Locking and Destruction in Different Tasks

```
#include <pthread.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;
pthread_mutex_t lock3;

void t0 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
  pthread_mutex_lock (&lock3);
  pthread_mutex_unlock (&lock2);
```

```
  pthread_mutex_unlock (&lock1);
  pthread_mutex_unlock (&lock3);
}

void t1 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
  pthread_mutex_destroy (&lock3);
  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}
```

In this example, after task `t0` locks the mutex `lock3`, task `t1` can destroy it. The destruction occurs if the following events happen in sequence:

1. `t0` acquires `lock3`.
2. `t0` releases `lock2`.
3. `t0` releases `lock1`.
4. `t1` acquires the lock `lock1` released by `t0`.
5. `t1` acquires the lock `lock2` released by `t0`.
6. `t1` destroys `lock3`.

For simplicity, this example uses a mix of automatic and manual concurrency detection. The tasks `t0` and `t1` are manually specified as entry points by using the option `Tasks (-entry-points)`. The critical sections are implemented through primitives `pthread_mutex_lock` and `pthread_mutex_unlock` that the software detects automatically. In practice, for entry point specification (thread creation), you will use primitives such as `pthread_create`. The next example shows how the defect can appear when you use `pthread_create`.

Correction — Place Lock-Unlock Pair Together in Same Critical Section as Destruction

The locking and destruction of `lock3` occurs inside the critical section imposed by `lock1` and `lock2`, but the unlocking occurs outside. One possible correction is to place the lock-unlock pair in the same critical section as the destruction of the mutex. Use one of these critical sections:

- Critical section imposed by `lock1` alone.
- Critical section imposed by `lock1` and `lock2`.

In this corrected code, the lock-unlock pair and the destruction is placed in the critical section imposed by `lock1` and `lock2`. When `t0` acquires `lock1` and `lock2`, `t1` has to wait for their release before it executes the instruction `pthread_mutex_destroy (&lock3);`. Therefore, `t1` cannot destroy mutex `lock3` in the locked state.

```
#include <pthread.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;
pthread_mutex_t lock3;

void t0 (void) {
  pthread_mutex_lock (&lock1);
```

```
  pthread_mutex_lock (&lock2);

  pthread_mutex_lock (&lock3);
  pthread_mutex_unlock (&lock3);

  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}

void t1 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);

  pthread_mutex_destroy (&lock3);

  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}
```

Example - Locking and Destruction in Start Routine of Thread

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 4
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
void atomic_operation(void);

void *do_create(void *arg) {
    /* Creation thread */
    pthread_mutex_init(&lock, NULL);
    pthread_exit((void*) 0);
}

void *do_work(void *arg) {
    /* Worker thread */
    pthread_mutex_lock (&lock);
    atomic_operation();
    pthread_mutex_unlock (&lock);
    pthread_exit((void*) 0);
}

void *do_destroy(void *arg) {
    /* Destruction thread */
    pthread_mutex_destroy(&lock);
    pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
```

```
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Thread that initializes mutex */
   pthread_create(&callThd[0], &attr, do_create, NULL);

   /* Threads that use mutex for atomic operation*/
   for(i=0; i<NUMTHREADS-1; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   /* Thread that destroys mutex */
   pthread_create(&callThd[NUMTHREADS -1], &attr, do_destroy, NULL);

   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   pthread_exit(NULL);
}
```

In this example, four threads are created. The threads are assigned different actions.

- The first thread `callThd[0]` initializes the mutex `lock`.
- The second and third threads, `callThd[1]` and `callThd[2]`, perform an atomic operation protected by the mutex `lock`.
- The fourth thread `callThd[3]` destroys the mutex `lock`.

The threads can interrupt each other. Therefore, immediately after the second or third thread locks the mutex, the fourth thread can destroy it.

Correction — Initialize and Destroy Mutex Outside Start Routine

One possible correction is to initialize and destroy the mutex in the `main` function outside the start routine of the threads. The threads perform only the atomic operation. You need two fewer threads because the mutex initialization and destruction threads are not required.

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 2
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
void atomic_operation(void);

void *do_work(void *arg) {
   pthread_mutex_lock (&lock);
   atomic_operation();
   pthread_mutex_unlock (&lock);
   pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   int i;
   void *status;
```

```
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Initialize mutex */
   pthread_mutex_init(&lock, NULL);

   for(i=0; i<NUMTHREADS; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* Destroy mutex */
   pthread_mutex_destroy(&lock);

   pthread_exit(NULL);
}
```

Correction — Use A Second Mutex To Protect Lock-Unlock Pair and Destruction

Another possible correction is to use a second mutex and protect the lock-unlock pair from the destruction. This corrected code uses the mutex `lock2` to achieve this protection. The second mutex is initialized in the `main` function outside the start routine of the threads.

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 4
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
pthread_mutex_t lock2;
void atomic_operation(void);

void *do_create(void *arg) {
    /* Creation thread */
    pthread_mutex_init(&lock, NULL);
    pthread_exit((void*) 0);
}

void *do_work(void *arg) {
    /* Worker thread */
    pthread_mutex_lock (&lock2);
    pthread_mutex_lock (&lock);
    atomic_operation();
    pthread_mutex_unlock (&lock);
    pthread_mutex_unlock (&lock2);
    pthread_exit((void*) 0);
}
```

```
void *do_destroy(void *arg) {
    /* Destruction thread */
    pthread_mutex_lock (&lock2);
    pthread_mutex_destroy(&lock);
    pthread_mutex_unlock (&lock2);
    pthread_exit((void*) 0);
}


int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Initialize second mutex */
   pthread_mutex_init(&lock2, NULL);

   /* Thread that initializes first mutex */
   pthread_create(&callThd[0], &attr, do_create, NULL);

   /* Threads that use first mutex for atomic operation */
   /* The threads use second mutex to protect first from destruction in locked state*/
   for(i=0; i<NUMTHREADS-1; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   /* Thread that destroys first mutex */
   /* The thread uses the second mutex to prevent destruction of locked mutex */
   pthread_create(&callThd[NUMTHREADS -1], &attr, do_destroy, NULL);


   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* Destroy second mutex */
   pthread_mutex_destroy(&lock2);

   pthread_exit(NULL);
}
```

Check Information

Group: Rule 14. Concurrency (CON)

See Also

Check SEI CERT-C (-cert-c)

Topics
“Check for Coding Standard Violations”

External Websites
CON31-C

Introduced in R2019a

CERT C: Rule CON32-C

Prevent data races when accessing bit fields from multiple threads

Description

Rule Definition

Prevent data races when accessing bit fields from multiple threads.

Polyspace Implementation

This checker checks for **Data race on adjacent bit fields**.

Examples

Data race on adjacent bit fields

Issue

Data race on adjacent bit fields occurs when:

1. Multiple tasks perform unprotected operations on bit fields that are part of the same structure.

   For instance, a task operates on field `errorFlag1` and another task on field `errorFlag2` in a variable of this type:

   ```
   struct errorFlags {
      unsigned int errorFlag1 : 1;
      unsigned int errorFlag2 : 1;
      ...
   }
   ```

   Suppose that the operations are not atomic with respect to each other. In other words, you have not implemented protection mechanisms to ensure that one operation completes before another begins.
2. At least one of the unprotected operations is a write operation.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

Risk

Adjacent bit fields that are part of the same structure might be stored in one byte in the same memory location. Read or write operations on all variables including bit fields happen one byte or word at a time. To modify only specific bits in a byte, steps similar to this happen in sequence:

1. The byte is loaded into RAM.
2. A mask is created so that only specific bits would be modified to the intended value and the remaining bits remain unchanged.
3. A bitwise OR operation is performed between the copy of the byte in RAM and the mask.

4 The byte with specific bits modified is copied back from RAM.

If two different bit fields are accessed, these four steps have to be performed for each bit field. If the accesses are not protected, all four steps for one bit field might not complete before the four steps for the other begin. As a result, the modification of one bit field might undo the modification of an adjacent bit field. For instance, the modification of `errorFlag1` and `errorFlag2` can happen in the following sequence. Steps marked 1 relate to modification of `errorFlag1` and steps marked 2 relate to that of `errorFlag2`.

1a. The byte with both `errorFlag1` and `errorFlag2` unmodified is copied into RAM, for purposes of modifying `errorFlag1`.

1b. A mask that modifies only `errorFlag1` is bitwise OR-ed with this copy.

2a. The byte containing both `errorFlag1` and `errorFlag2` unmodified is copied into RAM a second time, for purposes of modifying `errorFlag2`.

2b. A mask that modifies only `errorFlag2` is bitwise OR-ed with this second copy.

1c. The version with `errorFlag1` modified is copied back. This version has `errorFlag2` unmodified.

2c The version with `errorFlag2` modified is copied back. This version has `errorFlag1` unmodified and overwrites the previous modification.

Fix

To fix this defect, protect the operations on bit fields that are part of the same structure using critical sections, temporal exclusion or another means. See “Protections for Shared Variables in Multitasking Code”.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the  icon. For an example, see below.

Example - Unprotected Operation on Global Variable from Multiple Tasks

```
typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void task1 (void) {
    InterruptConfigbitsProc12.IOFlag = 0;
}

void task2 (void) {
```

```
    InterruptConfigbitsProc12.SetupFlag = 0;
}
```

In this example, `task1` and `task2` access different bit fields `IOFlag` and `SetupFlag`, which belong to the same structured variable `InterruptConfigbitsProc12`.

To emulate multitasking behavior, specify the following options:

Option	Specification
Configure multitasking manually on page 1-104	☑

Tasks on page 1-108	`task1`
	
	`task2`

On the command-line, you can use the following:

```
 polyspace-bug-finder
   -entry-points task1,task2
```

Correction – Use Critical Sections

One possible correction is to wrap the bit field accesses in a critical section. A critical section lies between a call to a lock function and an unlock function. In this correction, the critical section lies between the calls to functions `begin_critical_section` and `end_critical_section`.

```
typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void begin_critical_section(void);
void end_critical_section(void);

void task1 (void) {
    begin_critical_section();
    InterruptConfigbitsProc12.IOFlag = 0;
    end_critical_section();
}

void task2 (void) {
    begin_critical_section();
    InterruptConfigbitsProc12.SetupFlag = 0;
    end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

Option	Specification	
Configure multitasking manually on page 1-104	☑	

Tasks on page 1-108	`task1`
	
	`task2`	

Critical section details on page 1-119	**Starting routine**	**Ending routine**
	`begin_critical_section`	`end_critical_section`

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points task1,task2
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

Correction – Avoid Bit Fields

If you do not have memory constraints, use the `char` data type instead of bit fields. `char` variables in a structure occupy at least one byte and do not have the thread safety issues that come from bit manipulations in a byte-sized operation. Data races do not result from unprotected operations on different `char` variables that are part of the same structure.

```
typedef struct
{
   unsigned char IOFlag;
   unsigned char InterruptFlag;
   unsigned char Register1Flag;
   unsigned char SignFlag;
   unsigned char SetupFlag;
   unsigned char Register2Flag;
   unsigned char ProcessorFlag;
   unsigned char GeneralFlag;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void task1 (void) {
    InterruptConfigbitsProc12.IOFlag = 0;
}

void task2 (void) {
    InterruptConfigbitsProc12.SetupFlag = 0;
}
```

Though the checker does not flag this correction, do not use this correction for C99 or earlier. Only from C11 onwards does the C Standard mandate that distinct `char` variables cannot be accessed using the same word.

Correction – Insert Bit Field of Size 0

You can enter a non bit field member or an unnamed bit field member of size 0 between two adjacent bit fields that might be accessed concurrently. A non bit field member or size 0 bit field member ensures that the subsequent bit field starts from a new memory location. In this corrected example, the size 0 bit field member ensures that `IOFlag` and `SetupFlag` are stored in distinct memory locations.

```
typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int : 0;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void task1 (void) {
    InterruptConfigbitsProc12.IOFlag = 0;
}

void task2 (void) {
    InterruptConfigbitsProc12.SetupFlag = 0;
}
```

Check Information

Group: Rule 14. Concurrency (CON)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”
“Analyze Multitasking Programs in Polyspace”
“Protections for Shared Variables in Multitasking Code”

External Websites

CON32-C

Introduced in R2019a

CERT C: Rule CON33-C

Avoid race conditions when using library functions

Description

Rule Definition

Avoid race conditions when using library functions.

Polyspace Implementation

This checker checks for **Data race through standard library function call**.

Examples

Data race through standard library function call

Issue

Data race through standard library function call occurs when:

1. Multiple tasks call the same standard library function.

   For instance, multiple tasks call the `strerror` function.
2. The calls are not protected using a common protection.

   For instance, the calls are not protected by the same critical section.

Functions flagged by this defect are not guaranteed to be reentrant. A function is reentrant if it can be interrupted and safely called again before its previous invocation completes execution. If a function is not reentrant, multiple tasks calling the function without protection can cause concurrency issues. For the list of functions that are flagged, see CON33-C: Avoid race conditions when using library functions.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

Risk

The functions flagged by this defect are nonreentrant because their implementations can use global or static variables. When multiple tasks call the function without protection, the function call from one task can interfere with the call from another task. The two invocations of the function can concurrently access the global or static variables and cause unpredictable results.

The calls can also cause more serious security vulnerabilities, such as abnormal termination, denial-of-service attack, and data integrity violations.

Fix

To fix this defect, do one of the following:

- Use a reentrant version of the standard library function if it exists.

  For instance, instead of `strerror()`, use `strerror_r()` or `strerror_s()`. For alternatives to functions flagged by this defect, see the documentation for CON33-C.

- Protect the function calls using common critical sections or temporal exclusion.

  See `Critical section details (-critical-section-begin -critical-section-end)` and `Temporally exclusive tasks (-temporal-exclusions-file)`.

  To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the  icon. For an example, see below.

Example - Unprotected Call to Standard Library Function from Multiple Tasks

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno);
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

Option	Specification	
`Configure multitasking manually`	☑	

`Tasks (-entry-points)`	`task1`
	`task2`
	`task3`	

`Critical section details (-critical-section-begin -critical-section-end)`	**Starting routine**	**Ending routine**
	`begin_critical_section`	`end_critical_section`

On the command-line, you can use the following:

```
polyspace-bug-finder
  -entry-points task1,task2,task3
  -critical-section-begin begin_critical_section:cs1
  -critical-section-end end_critical_section:cs1
```

In this example, the tasks, `task1`, `task2` and `task3`, call the function `func`. `func` calls the nonreentrant standard library function, `strerror`.

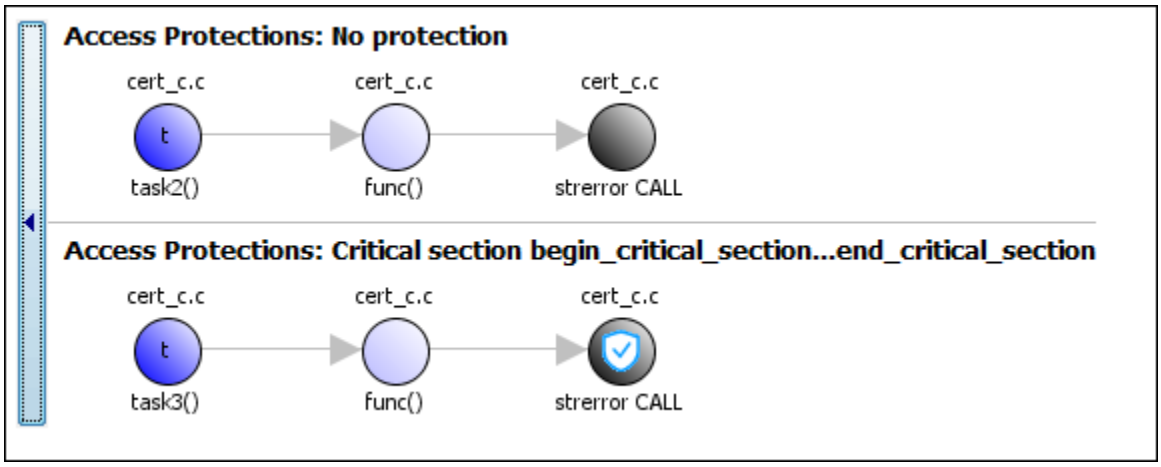
Though `task3` calls `func` inside a critical section, other tasks do not use the same critical section. Operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

These three tasks are calling a nonreentrant standard library function without common protection. In your result details, you see each pair of conflicting function calls.

! Data race through standard library function call (Impact: High)
Certain calls to function 'strerror' can interfere with each other and cause unpredictable results.
To avoid interference, calls to 'strerror' must be in the same critical section.

	Access	Access Protections	Task	File	Scope	Line
	Function call (Non atomic) Operation involves function call	No protection	task1()	data_race_std_lib.c	func()	14
	Function call (Non atomic) Operation involves function call	No protection	task2()	data_race_std_lib.c	func()	14
	Function call (Non atomic) Operation involves function call	No protection	task2()	data_race_std_lib.c	func()	14
	Function call (Non atomic) Operation involves function call	**Critical section begin_critical_section...end_critical_section**	task3()	data_race_std_lib.c	func()	14
	Function call (Non atomic) Operation involves function call	No protection	task1()	data_race_std_lib.c	func()	14
	Function call (Non atomic) Operation involves function call	**Critical section begin_critical_section...end_critical_section**	task3()	data_race_std_lib.c	func()	14

If you click the  icon, you see the function call sequence starting from the entry point to the standard library function call. You also see that the call starting from `task3` is in a critical section. The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.



Correction — Use Reentrant Version of Standard Library Function

One possible correction is to use a reentrant version of the standard library function `strerror`. You can use the POSIX version `strerror_r` which has the same functionality but also guarantees thread-safety.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);
enum { BUFFERSIZE = 64 };

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char errmsg[BUFFERSIZE];
    if (strerror_r(errno, errmsg, BUFFERSIZE) != 0) {
      /* Handle error */
    }
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
```

```
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

Correction — Place Function Call in Critical Section

One possible correction is to place the call to `strerror` in critical section. You can implement the critical section in multiple ways.

For instance, you can place the call to the intermediate function `func` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `func` and therefore the calls to `strerror` from the three tasks cannot interfere with each other.

To implement the critical section, in each of the three tasks, call `func` between calls to `begin_critical_section` and `end_critical_section`.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno);
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    begin_critical_section();
    func(fptr1);
    end_critical_section();
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     begin_critical_section();
     func(fptr2);
     end_critical_section();
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

Option	Value
`Temporally exclusive tasks (-temporal-exclusions-file)`	`task1 task2 task3`

On the command-line, you can use the following:

```
polyspace-bug-finder
    -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

Check Information

Group: Rule 14. Concurrency (CON)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

CON33-C

Introduced in R2019a

CERT C: Rule CON34-C

Declare objects shared between threads with appropriate storage durations

Description

Rule Definition

Declare objects shared between threads with appropriate storage durations.

Examples

Automatic or thread local variable escaping from a C11 thread

Issue

Automatic or thread local variable escaping from a C11 thread occurs when an automatic or thread local variable is passed by address from one C11 thread to another without ensuring that the variable stays alive through the duration of the latter thread.

Risk

An automatic or thread local variable is allocated on the stack at the beginning of a thread and its lifetime extends till the end of the thread. The variable is not guaranteed to be alive when a different thread accesses it.

For instance, consider the start function of a C11 thread with these lines:

```
int start_thread(thrd_t *tid) {
   int aVar = 0;
   if(thrd_success != thrd_create(tid, start_thread_child, &aVar) {
     ...
   }
}
```

The `thrd_create` function creates a child thread with start function `start_thread_child` and passes the address of the automatic variable `aVar`to this function. When this child thread accesses `aVar`, the parent thread might have completed execution and `aVar` is no longer on the stack. The access might result in reading unpredictable values.

Fix

When you pass a variable from one thread to another, make sure that the variable lifetime matches or exceeds the lifetime of both threads. You can achieve this synchronization in one of these ways:

- Declare the variable `static` so that it does not go out of stack when the current thread completes execution.
- Dynamically allocate the storage for the variable so that it is allocated on the heap instead of the stack and must be explicitly deallocated. Make sure that the deallocation happens after both threads complete execution.

These solutions require you to create a variable in nonlocal memory. Instead, you can use other solutions such as the `shared` keyword available with OpenMP's threading interface that allows you to safely share local variables across threads.

Example – Automatic or Thread-Local Variable Escaping Thread

```
#include <threads.h>
#include <stdio.h>

int create_child_thread(void *childVal) {
  int *res = (int *)childVal;
  printf("Result: %d\n", *res);
  return 0;
}

void create_parent_thread(thrd_t *tid, int *parentPtr) {
   if (thrd_success != thrd_create(tid, create_child_thread, parentPtr)) {
    /* Handle error */
  }
}

int main(void) {
  thrd_t tid;
  int parentVal = 1;

  create_parent_thread(&tid, &parentVal);


  if (thrd_success != thrd_join(tid, NULL)) {
    /* Handle error */
  }
  return 0;
}
```

In this example, the value `parentVal` is local to the parent thread that starts in `main` and continues into the function `create_parent_thread`. However, in the body of `create_parent_thread`, the address of this local variable is passed to a child thread (the thread with start routine `create_child_thread`). The parent thread might have completed execution and the variable `parentVal` might have gone out of scope when the child thread accesses this variable.

The same issue appears if the variable is declared as thread-local, for instance with the C11 keyword `_Thread_local` (or `thread_local`):

```
_Thread_local int parentVal = 1;
```

Correction – Use Static Variables

One possible correction is to declare the variable `parentVal` as `static` so that the variable is on the stack for the entire duration of the program.

```
#include <threads.h>
#include <stdio.h>

int create_child_thread(void *childVal) {
  int *res = (int *)childVal;
  printf("Result: %d\n", *res);
  return 0;
}
```

```
void create_parent_thread(thrd_t *tid, int *parentPtr) {
   if (thrd_success != thrd_create(tid, create_child_thread, parentPtr)) {
    /* Handle error */
  }
}

int main(void) {
  thrd_t tid;
  static int parentVal = 1;

  create_parent_thread(&tid, &parentVal);


  if (thrd_success != thrd_join(tid, NULL)) {
    /* Handle error */
  }
  return 0;
}
```

Correction – Use Dynamic Memory Allocation

One possible correction is to dynamically allocate storage for variables to be shared across threads and explicitly free the storage after the variable is no longer required.

```
#include <threads.h>
#include <stdio.h>

int create_child_thread(void *childVal) {
  int *res = (int *)childVal;
  printf("Result: %d\n", *res);
  return 0;
}

void create_parent_thread(thrd_t *tid, int *parentPtr) {
   if (thrd_success != thrd_create(tid, create_child_thread, parentPtr)) {
    /* Handle error */
  }
}

int main(void) {
  thrd_t tid;
  int parentPtr = (int*) malloc(sizeof(int));

  if(!parentPtr) {
      create_parent_thread(&tid, &parentVal);


      if (thrd_success != thrd_join(tid, NULL)) {
        /* Handle error */
      }
      free(parentPtr);
  }
  return 0;
}
```

Check Information

Group: Rule 14. Concurrency (CON)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

CON34-C

Introduced in R2020a

CERT C: Rule CON35-C

Avoid deadlock by locking in a predefined order

Description

Rule Definition

Avoid deadlock by locking in a predefined order.

Polyspace Implementation

This checker checks for **Deadlock**.

Examples

Deadlock

Issue

Deadlock occurs when multiple tasks are stuck in their critical sections (CS) because:

- Each CS waits for another CS to end.
- The critical sections (CS) form a closed cycle. For example:
  - CS #1 waits for CS #2 to end, and CS #2 waits for CS #1 to end.
  - CS #1 waits for CS #2 to end, CS #2 waits for CS #3 to end and CS #3 waits for CS #1 to end.

Polyspace expects critical sections of code to follow a specific format. A critical section lies between a call to a lock function and a call to an unlock function. When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait until `my_task` calls the corresponding unlock function. Both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Risk

Each task waits for a critical section in another task to end and is unable to proceed. The program can freeze indefinitely.

Fix

The fix depends on the root cause of the defect. You can try to break the cyclic order between the tasks in one of these ways:

- Write down all critical sections involved in the deadlock in a certain sequence. Whenever you call the lock functions of the critical sections within a task, respect the order in that sequence. See an example below.
- If one of the critical sections involved in a deadlock occurs in an interrupt, try to disable all interrupts during critical sections in all tasks. See `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Reviewing this defect is an opportunity to check if all operations in your critical section are really meant to be executed as an atomic block. It is a good practice to keep critical sections at a bare minimum.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Deadlock with Two Tasks

```
void task1(void);
void task2(void);

int var;
void perform_task_cycle(void) {
 var++;
}

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_2();
    begin_critical_section_1();
    perform_task_cycle();
    end_critical_section_1();
    end_critical_section_2();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Specification
Configure multitasking manually	☑

Entry points	`task1`
	
	`task2`

Option	Specification	
Critical section details	**Starting routine**	**Ending routine**
	`begin_critical_section_1`	`end_critical_section_1`
	`begin_critical_section_2`	`end_critical_section_2`

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. `task1` calls `begin_critical_section_1`.
2. `task2` calls `begin_critical_section_2`.
3. `task1` reaches the instruction `begin_critical_section_2();`. Since `task2` has already called `begin_critical_section_2`, `task1` waits for `task2` to call `end_critical_section_2`.
4. `task2` reaches the instruction `begin_critical_section_1();`. Since `task1` has already called `begin_critical_section_1`, `task2` waits for `task1` to call `end_critical_section_1`.

Correction-Follow Same Locking Sequence in Both Tasks

One possible correction is to follow the same sequence of calls to lock and unlock functions in both `task1` and `task2`.

```
void task1(void);
void task2(void);
void perform_task_cycle(void);

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}
```

Example - Deadlock with More Than Two Tasks

```
int var;
void performTaskCycle() {
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);


void unlock1(void);
void unlock2(void);
void unlock3(void);

void task1() {
 while(1) {
    lock1();
    lock2();
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock3();
    lock1();
    performTaskCycle();
    unlock1();
    unlock3();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Specification
Configure multitasking manually	☑

Option	Specification	

Entry points	`task1`
	`task2`
	`task3`	

Critical section details	**Starting routine**	**Ending routine**
	`lock1`	`unlock1`
	`lock2`	`unlock2`
	`lock3`	`unlock3`

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. `task1` calls `lock1`.
2. `task2` calls `lock2`.
3. `task3` calls `lock3`.
4. `task1` reaches the instruction `lock2();`. Since `task2` has already called `lock2`, `task1` waits for call to `unlock2`.
5. `task2` reaches the instruction `lock3();`. Since `task3` has already called `lock3`, `task2` waits for call to `unlock3`.
6. `task3` reaches the instruction `lock1();`. Since `task1` has already called `lock1`, `task3` waits for call to `unlock1`.

Correction — Break Cyclic Order

To break the cyclic order between critical sections, note every lock function in your code in a certain sequence, for example:

1. `lock1`
2. `lock2`
3. `lock3`

If you use more than one lock function in a task, use them in the order in which they appear in the sequence. For example, you can use `lock1` followed by `lock2` but not `lock2` followed by `lock1`.

```
int var;
void performTaskCycle() {
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);

void unlock1(void);
void unlock2(void);
void unlock3(void);
```

```
void task1() {
 while(1) {
    lock1();
    lock2();
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock1();
    lock3();
    performTaskCycle();
    unlock3();
    unlock1();
 }
}
```

Check Information

Group: Rule 14. Concurrency (CON)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

CON35-C

Introduced in R2019a

CERT C: Rule CON36-C

Wrap functions that can spuriously wake up in a loop

Description

Rule Definition

Wrap functions that can spuriously wake up in a loop.

Polyspace Implementation

This checker checks for **Function that can spuriously wake up not wrapped in loop**.

Examples

Function that can spuriously wake up not wrapped in loop

Issue

Function that can spuriously wake up not wrapped in loop occurs when the following wait-on-condition functions are called from outside a loop:

- C functions:
  - `cnd_wait()`
  - `cnd_timedwait()`
- POSIX functions:
  - `pthread_cond_wait()`
  - `pthread_cond_timedwait()`
- C++ `std::condition_variable` and `std::condition_variable_any` class member functions:
  - `wait()`
  - `wait_until()`
  - `wait_for()`

Wait-on-condition functions pause the execution of the calling thread when a specified condition is met. The thread wakes up and resumes once another thread notifies it with `cnd_broadcast()` or an equivalent function. The wake-up notification can be spurious or malicious.

Risk

If a thread receives a spurious wake-up notification and the condition of the wait-on-condition function is not checked, the thread can wake up prematurely. The wake-up can cause unexpected control flow, indefinite blocking of other threads, or denial of service.

Fix

Wrap wait-on-condition functions that can wake up spuriously in a loop. The loop checks the wake-up condition after a possible spurious wake-up notification.

Example - `cnd_wait()` Not Wrapped in Loop

```
#include <stdio.h>
#include <stddef.h>
#include <threads.h>

#define THRESHOLD 100

static mtx_t lock;
static cnd_t cond;

void func(int input)
{
    if (thrd_success != mtx_lock(&lock)) {
        /* Handle error */
    }
    /* test condition to pause thread */
    if (input > THRESHOLD) {
        if (thrd_success != cnd_wait(&cond, &lock)) {
            /* Handle error */
        }
    }
    /* Proceed if condition to pause does not hold */


    if (thrd_success != mtx_unlock(&lock)) {
        /* Handle error */
    }
}
```

In this example, the thread uses `cnd_wait()` to pause execution when `input` is greater than `THRESHOLD`. The paused thread can resume if another thread uses `cnd_broadcast()`, which notifies all the threads. This notification causes the thread to wake up even if the pause condition is still true.

Correction — Wrap `cnd_wait()` in a `while` Loop

One possible correction is to wrap `cnd_wait()` in a `while` loop. The loop checks the pause condition after the thread receives a possible spurious wake-up notification.

```
#include <stdio.h>
#include <stddef.h>
#include <threads.h>

#define THRESHOLD 100

static mtx_t lock;
static cnd_t cond;

void func(int input)
{
    if (thrd_success != mtx_lock(&lock)) {
        /* Handle error */
    }
    /* test condition to pause thread */
    while (input > THRESHOLD) {
        if (thrd_success != cnd_wait(&cond, &lock)) {
            /* Handle error */
```

```
        }
    }
    /* Proceed if condition to pause does not hold */


    if (thrd_success != mtx_unlock(&lock)) {
        /* Handle error */
    }
}
```

Check Information

Group: Rule 14. Concurrency (CON)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

CON36-C

Introduced in R2019a

CERT C: Rule CON37-C

Do not call signal() in a multithreaded program

Description

Rule Definition

Do not call signal() in a multithreaded program.

Polyspace Implementation

This checker checks for **Signal call in multithreaded program**.

Examples

Signal call in multithreaded program

Issue

Signal call in multithreaded program occurs when you use the `signal()` function in a program with multiple threads.

Risk

According to the C11 standard (Section 7.14.1.1), use of the `signal()` function in a multithreaded program is undefined behavior.

Fix

Depending on your intent, use other ways to perform an asynchronous action on a specific thread.

Example - Use of `signal()` Function to Terminate Loop in Thread

```
#include <signal.h>
#include <stddef.h>
#include <threads.h>

volatile sig_atomic_t flag = 0;

void handler(int signum) {
  flag = 1;
}

/* Runs until user sends SIGUSR1 */
int func(void *data) {
  while (!flag) {
    /* ... */
  }
  return 0;
}

int main(void) {
  signal(SIGINT, handler); /* Undefined behavior */
  thrd_t tid;
```

```
  if (thrd_success != thrd_create(&tid, func, NULL)) {
    /* Handle error */
  }
  /* ... */
  return 0;
}
```

In this example, the `signal` function is used to terminate a `while` loop in the thread created with `thrd_create`.

Correction — Use `atomic_bool` Variable to Terminate Loop

One possible correction is to use an `atomic_bool` variable that multiple threads can access. In the corrected example, the child thread evaluates this variable before every loop iteration. After completing the program, you can modify this variable so that the child thread exits the loop.

```
#include <stdatomic.h>
#include <stdbool.h>
#include <stddef.h>
#include <threads.h>

atomic_bool flag = ATOMIC_VAR_INIT(false);

int func(void *data) {
  while (!flag) {
    /* ... */
  }
  return 0;
}

int main(void) {
  thrd_t tid;

  if (thrd_success != thrd_create(&tid, func, NULL)) {
    /* Handle error */
  }
  /* ... */
  /* Set flag when done */
  flag = true;

  return 0;
}
```

Check Information

Group: Rule 14. Concurrency (CON)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

CON37-C

Introduced in R2019a

CERT C: Rule CON38-C

Preserve thread safety and liveness when using condition variables

Description

Rule Definition

Preserve thread safety and liveness when using condition variables.

Polyspace Implementation

This checker checks for **Multiple threads waiting on same condition variable**.

Examples

Multiple threads waiting on same condition variable

Issue

This issue occurs when you use `cnd_signal` family functions to wake up one of at least two threads that are concurrently waiting on the same condition variable. For threads with the same priority level, `cnd_signal` family functions cause the thread scheduler to arbitrarily wake up on of the threads waiting on the condition variable that you signal with the `cnd_signal` family function.

The checkers flags the `cnd_signal` family function call. See the **Event** column in the **Results Details** pane to view the threads waiting on the same condition variable.

Risk

The thread that is woken up with a `cnd_signal` family function usually tests for a condition predicate. While the condition predicate is false, the thread waits again on the condition variable until it is woken up by another thread that signals the condition variable. It is possible that the program ends up in a state where no thread is available to signal the condition variable, which results in indefinite blocking.

Fix

Use `cnd_broadcast` family functions instead to wake all threads waiting on the condition variable, or use a different condition variable for each thread.

Example - Use of `cnd_signal` to Wake Up One of Many Threads Waiting on Condition Variable

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <threads.h>

typedef int thrd_return_t;

static void fatal_error(void)
{
    exit(1);
}
```

```
enum { NTHREADS = 5 };

mtx_t mutex;
cnd_t cond;

thrd_return_t next_step(void* t)
{
    static size_t current_step = 0;
    size_t my_step = *(size_t*)t;

    if (thrd_success != mtx_lock(&mutex)) {
        /* Handle error */
        fatal_error();
    }

    printf("Thread %zu has the lock\n", my_step);
    while (current_step != my_step) {
        printf("Thread %zu is sleeping...\n", my_step);
        if (thrd_success !=
            cnd_wait(&cond, &mutex)) {
            /* Handle error */
            fatal_error();
        }
        printf("Thread %zu woke up\n", my_step);
    }
    /* Do processing ... */
    printf("Thread %zu is processing...\n", my_step);
    current_step++;

    /* Signal a waiting task */
    if (thrd_success !=
        cnd_signal(&cond)) {
        /* Handle error */
        fatal_error();
    }

    printf("Thread %zu is exiting...\n", my_step);

    if (thrd_success != mtx_unlock(&mutex)) {
        /* Handle error */
        fatal_error();
    }
    return (thrd_return_t)0;
}

int main(void)
{
    thrd_t threads[NTHREADS];
    size_t step[NTHREADS];

    if (thrd_success != mtx_init(&mutex, mtx_plain)) {
        /* Handle error */
        fatal_error();
    }
    if (thrd_success != cnd_init(&cond)) {
        /* Handle error */
        fatal_error();
```

```
    }
    /* Create threads */
    for (size_t i = 0; i < NTHREADS; ++i) {
        step[i] = i;
        if (thrd_success != thrd_create(&threads[i],
                                        next_step,
                                        &step[i])) {
            /* Handle error */
            fatal_error();
        }
    }
    /* Wait for all threads to complete */
    for (size_t i = NTHREADS; i != 0; --i) {
        if (thrd_success != thrd_join(threads[i - 1], NULL)) {
            /* Handle error */
            fatal_error();
        }
    }
    (void)mtx_destroy(&mutex);
    (void)cnd_destroy(&cond);
    return 0;
}
```

In this example, multiple threads are created and assigned step level. Each thread checks if its assigned step level matches the current step level (condition predicate). If the predicate is false, the thread goes back to waiting on the condition variable `cond`. The use of `cnd_signal` to signal the `cond` causes the thread scheduler to arbitrarily wake up one of the threads waiting on `cond`. This can result in indefinite blocking when the condition predicate of woken up thread is false and no other thread is available to signal `cond`.

Correction — Use `cnd_broadcast` to Wake up All the Threads

One possible correction is to use `cnd_broadcast` instead to signal `cond`. The function `cnd_signal` wakes up all the thread that are waiting on `cond`.

```
 #include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <threads.h>

typedef int thrd_return_t;

static void fatal_error(void)
{
    exit(1);
}

enum { NTHREADS = 5 };

mtx_t mutex;
cnd_t cond;

thrd_return_t next_step(void* t)
{
    static size_t current_step = 0;
    size_t my_step = *(size_t*)t;

    if (thrd_success != mtx_lock(&mutex)) {
```

```
        /* Handle error */
        fatal_error();
    }

    printf("Thread %zu has the lock\n", my_step);
    while (current_step != my_step) {
        printf("Thread %zu is sleeping...\n", my_step);
        if (thrd_success !=
            cnd_wait(&cond, &mutex)) {
            /* Handle error */
            fatal_error();
        }
        printf("Thread %zu woke up\n", my_step);
    }
    /* Do processing ... */
    printf("Thread %zu is processing...\n", my_step);
    current_step++;

    /* Signal a waiting task */
    if (thrd_success !=
        cnd_broadcast(&cond)) {
        /* Handle error */
        fatal_error();
    }

    printf("Thread %zu is exiting...\n", my_step);

    if (thrd_success != mtx_unlock(&mutex)) {
        /* Handle error */
        fatal_error();
    }
    return (thrd_return_t)0;
}

int main_test_next_step(void)
{
    thrd_t threads[NTHREADS];
    size_t step[NTHREADS];

    if (thrd_success != mtx_init(&mutex, mtx_plain)) {
        /* Handle error */
        fatal_error();
    }
    if (thrd_success != cnd_init(&cond)) {
        /* Handle error */
        fatal_error();
    }
    /* Create threads */
    for (size_t i = 0; i < NTHREADS; ++i) {
        step[i] = i;
        if (thrd_success != thrd_create(&threads[i],
                                        next_step,
                                        &step[i])) {
            /* Handle error */
            fatal_error();
        }
    }
    /* Wait for all threads to complete */
```

```
    for (size_t i = NTHREADS; i != 0; --i) {
        if (thrd_success != thrd_join(threads[i - 1], NULL)) {
            /* Handle error */
            fatal_error();
        }
    }
    (void)mtx_destroy(&mutex);
    (void)cnd_destroy(&cond);
    return 0;
}
```

Check Information

Group: Rule 14. Concurrency (CON)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

CON38-C

Introduced in R2020a

CERT C: Rule CON39-C

Do not join or detach a thread that was previously joined or detached

Description

Rule Definition

Do not perform operations that can block while holding a lock.

Polyspace Implementation

This checker checks for **Join or detach of a joined or detached thread**.

Examples

Join or detach of a joined or detached thread

Issue

Join or detach of a joined or detached thread occurs when:

- A thread that is joined was previously joined or detached
- A thread that is detached was previously joined or detached.

The **Result Details** pane describes if the thread was previously joined or detached and also shows previous related events.

For instance, the issue occurs when a thread joined with `thrd_join` is then detached with `pthread_detach`:

```
thrd_t id;
...
thrd_join(id, NULL);
thrd_detach(id);
```

Note that a thread is considered as joined only if a previous thread joining is successful. For instance, the thread is not considered as joined in the `if` branch here:

```
thrd_t t;
...
if (thrd_success != thrd_join(t, 0)) {
    /* Thread not considered joined */
}
```

The analysis cannot detect cases where a joined thread detaches itself using, for instance, the `thrd_current()` function.

Risk

The C11 standard (clauses 7.26.5.3 and 7.26.5.6) states that a thread shall not be joined or detached once it was previously joined or detached. Violating these clauses of the standard results in undefined behavior.

Fix

Avoid joining a thread that was already joined or detached previously. Likewise, avoid detaching a thread that was already joined or detached.

Example – Joining Followed by Detaching of Thread

```
#include <stddef.h>
#include <threads.h>
#include <stdlib.h>


extern int thread_func(void *arg);

int main (void)
{
  thrd_t t;

  if (thrd_success != thrd_create (&t, thread_func, NULL)) {
    /* Handle error */
    return 0;
  }

  if (thrd_success != thrd_join (t, 0)) {
    /* Handle error */
    return 0;
  }

  if (thrd_success != thrd_detach (t)) {
    /* Handle error */
    return 0;
  }

  return 0;
}
```

In this example, the use of `thrd_detach` on a thread that was previously joined with `thrd_join` leads to undefined behavior.

To avoid compilation errors with this example, specify the C11 standard with the option `C standard version (-c-version)`.

Correction – Avoid Detaching a Joined Thread

Remove the `thrd_join` or `thrd_detach` statement.

```
#include <stddef.h>
#include <threads.h>
#include <stdlib.h>


extern int thread_func(void *arg);

int main (void)
{
  thrd_t t;
```

```
  if (thrd_success != thrd_create (&t, thread_func, NULL)) {
    /* Handle error */
    return 0;
  }

  if (thrd_success != thrd_join (t, 0)) {
    /* Handle error */
    return 0;
  }

  return 0;
}
```

Example – Joining Thread Created in Detached State

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);


int main() {
    pthread_t id;
    pthread_attr_t attr;

    if(thread_success != pthread_attr_init(&attr)) {
        return 0;
    }

    if(thread_success != pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED)) {
        return 0;
    }

    if(thread_success != pthread_create(&id, &attr, thread_func, NULL)) {
            return 0;
    }

    if(thread_success != pthread_join(id, NULL)) {
            return 0;
    }

    return 0;
}
```

In this example, the thread attribute is assigned the state `PTHREAD_CREATE_DETACHED`. A thread created using this attribute is then joined.

Correction – Create Threads as Joinable

One possible correction is to create a thread with thread attribute assigned to the state `PTHREAD_CREATE_JOINABLE` and then join the thread.

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);
```

```
int main() {
    pthread_t id;
    pthread_attr_t attr;

    if(thread_success != pthread_attr_init(&attr)) {
        return 0;
    }

    if(thread_success != pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE)) {
        return 0;
    }

    if(thread_success != pthread_create(&id, &attr, thread_func, NULL)) {
            return 0;
    }

    if(thread_success != pthread_join(id, NULL)) {
            return 0;
    }

    return 0;
}
```

Check Information

Group: Rule 14. Concurrency (CON)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

CON39-C

Introduced in R2019b

CERT C: Rule CON40-C

Do not refer to an atomic variable twice in an expression

Description

Rule Definition

Do not refer to an atomic variable twice in an expression.

Polyspace Implementation

This checker checks for these issues:

- **Atomic variable accessed twice in an expression**.
- **Atomic load and store sequence not atomic**.

Examples

Atomic variable accessed twice in an expression

Issue

Atomic variable accessed twice in an expression occurs when C atomic types or C++ `std::atomic` class variables appear twice in an expression and there are:

- Two atomic read operations on the variable.
- An atomic read and a distinct atomic write operation on the variable.

The C standard defines certain operations on atomic variables that are thread safe and do not cause data race conditions. Unlike individual operations, a pair of operations on the same atomic variable in an expression is not thread safe.

Risk

A thread can modify the atomic variable between the pair of atomic operations, which can result in a data race condition.

Fix

Do not reference an atomic variable twice in the same expression.

Example - Referencing Atomic Variable Twice in an Expression

```
#include <stdatomic.h>

atomic_int n = ATOMIC_VAR_INIT(0);

int compute_sum(void)
{
    return n * (n + 1) / 2;
}
```

In this example, the global variable `n` is referenced twice in the return statement of `compute_sum()`. The value of `n` can change between the two distinct read operations. `compute_sum()` can return an incorrect value.

Correction — Pass Variable as Function Argument

One possible correction is to pass the variable as a function argument `n`. The variable is copied to memory and the read operations on the copy guarantee that `compute_sum()` returns a correct result. If you pass a variable of type `int` instead of type `atomic_int`, the correction is still valid.

```
#include <stdatomic.h>

int compute_sum(atomic_int n)
{
    return n * (n + 1) / 2;
}
```

Atomic load and store sequence not atomic

Issue

Atomic load and store sequence not atomic occurs when you use these functions to load, and then store an atomic variable.

- C functions:
  - `atomic_load()`
  - `atomic_load_explicit()`
  - `atomic_store()`
  - `atomic_store_explicit()`
- C++ functions:
  - `std::atomic_load()`
  - `std::atomic_load_explicit()`
  - `std::atomic_store()`
  - `std::atomic_store_explicit()`
  - `std::atomic::load()`
  - `std::atomic::store()`

A thread cannot interrupt an atomic load or an atomic store operation on a variable, but a thread can interrupt a store, and then load sequence.

Risk

A thread can modify a variable between the load and store operations, resulting in a data race condition.

Fix

To read, modify, and store a variable atomically, use a compound assignment operator such as `+=`, `atomic_compare_exchange()` or `atomic_fetch_*`-family functions.

Example - Loading Then Storing an Atomic Variable

```
#include <stdatomic.h>
#include <stdbool.h>
```

```
static atomic_bool flag = ATOMIC_VAR_INIT(false);

void init_flag(void)
{
    atomic_init(&flag, false);
}

void toggle_flag(void)
{
    bool temp_flag = atomic_load(&flag);
    temp_flag = !temp_flag;
    atomic_store(&flag, temp_flag);
}

bool get_flag(void)
{
    return atomic_load(&flag);
}
```

In this example, variable `flag` of type `atomic_bool` is referenced twice inside the `toggle_flag()` function. The function loads the variable, negates its value, then stores the new value back to the variable. If two threads call `toggle_flag()`, the second thread can access `flag` between the load and store operations of the first thread. `flag` can end up in an incorrect state.

Correction — Use Compound Assignment to Modify Variable

One possible correction is to use a compound assignment operator to toggle the value of `flag`. The C standard defines the operation by using ^= as atomic.

```
 #include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void toggle_flag(void)
{
    flag ^= 1;
}

bool get_flag(void)
{
    return flag;
}
```

Check Information

Group: Rule 14. Concurrency (CON)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

CON40-C

Introduced in R2019a

CERT C: Rule CON41-C

Wrap functions that can fail spuriously in a loop

Description

Rule Definition

Wrap functions that can fail spuriously in a loop.

Polyspace Implementation

This checker checks for **Function that can spuriously fail not wrapped in loop**.

Examples

Function that can spuriously fail not wrapped in loop

Issue

Function that can spuriously fail not wrapped in loop occurs when the following atomic compare and exchange functions that can fail spuriously are called from outside a loop.

- C atomic functions:
  - `atomic_compare_exchange_weak()`
  - `atomic_compare_exchange_weak_explicit()`
- C++ atomic functions:
  - `std::atomic<T>::compare_exchange_weak(T* expected, T desired)`
  - `std::atomic<T>::compare_exchange_weak_explicit(T* expected, T desired, std::memory_order succ, std::memory_order fail)`
  - `std::atomic_compare_exchange_weak(std::atomic<T>* obj, T* expected, T desired)`
  - `std::atomic_compare_exchange_weak_explicit(volatile std::atomic<T>* obj, T* expected, T desired, std::memory_order succ, std::memory_order fail)`

The functions compare the memory contents of the object representations pointed to by `obj` and `expected`. The comparison can spuriously return false even if the memory contents are equal. This spurious failure makes the functions faster on some platforms.

Risk

An atomic compare and exchange function that spuriously fails can cause unexpected results and unexpected control flow.

Fix

Wrap atomic compare and exchange functions that can spuriously fail in a loop. The loop checks the failure condition after a possible spurious failure.

Example - `atomic_compare_exchange_weak()` Not Wrapped in Loop

```
#include <stdatomic.h>

extern void reset_count(void);
atomic_int count = ATOMIC_VAR_INIT(0);

void increment_count(void)
{
    int old_count = atomic_load(&count);
    int new_count;
    new_count = old_count + 1;
    if (!atomic_compare_exchange_weak(&count, &old_count, new_count))
        reset_count();

}
```

In this example, `increment_count()` uses `atomic_compare_exchange_weak()` to compare `count` and `old_count`. If the counts are equal, `count` is incremented to `new_count`. If they are not equal, the count is reset. When `atomic_compare_exchange_weak()` fails spuriously, the count is reset unnecessarily.

Correction — Wrap `atomic_compare_exchange_weak()` in a `while` Loop

One possible correction is to wrap the call to `atomic_compare_exchange_weak()` in a `while` loop. The loop checks the failure condition after a possible spurious failure.

```
#include <stdatomic.h>

extern void reset_count(void);
atomic_int count = ATOMIC_VAR_INIT(0);

void increment_count(void)
{
    int old_count = atomic_load(&count);
    int new_count;
    new_count = old_count + 1;

    do {
        reset_count();

    } while (!atomic_compare_exchange_weak(&count, &old_count, new_count));

}
```

Check Information

Group: Rule 14. Concurrency (CON)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

CON41-C

Introduced in R2019a

CERT C: Rule CON43-C

Do not allow data races in multithreaded code

Description

Rule Definition

Do not allow data races in multithreaded code.

Polyspace Implementation

This checker checks for **Data race**.

Examples

Data race

Issue

Data race occurs when:

1. Multiple tasks perform unprotected operations on a shared variable.
2. At least one task performs a write operation.
3. At least one operation is nonatomic. For data race on both atomic and nonatomic operations, see `Data race including atomic operations`.

   See also “Define Atomic Operations in Multitasking Code”.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

Risk

Data race can result in unpredictable values of the shared variable because you do not control the order of the operations in different tasks.

Data races between two write operations are more serious than data races between a write and read operation. Two write operations can interfere with each other and result in indeterminate values. To identify write-write conflicts, use the filters on the **Detail** column of the **Results List** pane. For these conflicts, the **Detail** column shows the additional line:

```
Variable value may be altered by write-write concurrent access.
```

See “Filter and Group Results”.

Fix

To fix this defect, protect the operations on the shared variable using critical sections, temporal exclusion or another means. See “Protections for Shared Variables in Multitasking Code”.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing

protections on the calls. To see the function call sequence leading to the conflicts, click the  icon. For an example, see below.

Example - Unprotected Operation on Global Variable from Multiple Tasks

```
int var;
void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
    var++;
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

Option	Specification	
Configure multitasking manually on page 1-104	☑	

Tasks on page 1-108	`task1`
	`task2`
	`task3`	

Critical section details on page 1-119	**Starting routine**	**Ending routine**
	`begin_critical_section`	`end_critical_section`

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points task1,task2,task3
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

In this example, the tasks `task1`, `task2`, and `task3` call the function `increment`. `increment` contains the operation `var++` that can involve multiple machine instructions including:

- Reading `var`.

- Writing an increased value to `var`.

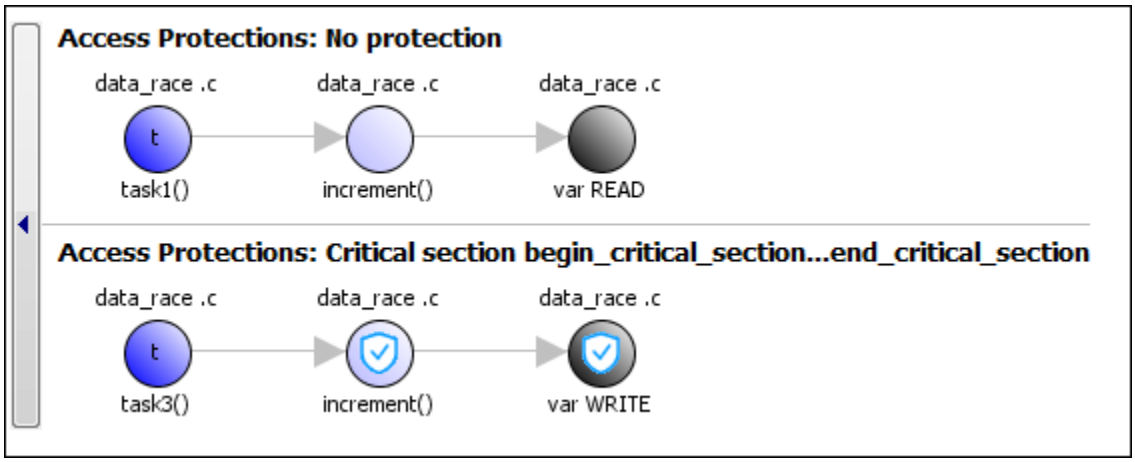
These machine instructions, when executed from `task1` and `task2`, can occur concurrently in an unpredictable sequence. For example, reading `var` from `task1` can occur either before or after writing to `var` from `task2`. Therefore the value of `var` can be unpredictable.

Though `task3` calls `increment` inside a critical section, other tasks do not use the same critical section. The operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

Therefore, the three tasks are operating on a shared variable without common protection. In your result details, you see each pair of conflicting function calls.

Access	Access Protections	Task	File
Read	No protection	task1()	data_race .c
Write (Non atomic) Operation might involve multiple machine instructions	No protection	task2()	data_race .c
Read	No protection	task1()	data_race .c
Write (Non atomic) Operation might involve multiple machine instructions	**Critical section begin_critical_section...end_critical_section**	task3()	data_race .c
Read	No protection	task2()	data_race .c
Write (Non atomic) Operation might involve multiple machine instructions	**Critical section begin_critical_section...end_critical_section**	task3()	data_race .c

If you click the  icon, you see the function call sequence starting from the entry point to the read or write operation. You also see that the operation starting from `task3` is in a critical section. The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.



Correction — Place Operation in Critical Section

One possible correction is to place the operation in critical section. You can implement the critical section in multiple ways. For instance:

- You can place `var++` in a critical section. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The operation `var++` from the three tasks cannot interfere with each other.

To implement the critical section, in the function `increment`, place the operation `var++` between calls to `begin_critical_section` and `end_critical_section`.

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
      begin_critical_section();
      var++;
      end_critical_section();
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
      increment();
}
```

- You can place the call to `increment` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `increment` from the three tasks cannot interfere with each other.

  To implement the critical section, in each of the three tasks, call `increment` between calls to `begin_critical_section` and `end_critical_section`.

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
      var++;
}

void task1(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}

void task2(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
```

```
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

Option	Value
Temporally exclusive tasks on page 1-122	`task1 task2 task3`

On the command-line, you can use the following:

```
polyspace-bug-finder
    -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

Example - Unprotected Operation in Threads Created with `pthread_create`

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;


void* increment_count(void* args)
{
    count = count + 1;
    return NULL;
}

void* set_count(void *args)
{
    long long c;
    c = count;
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);

    pthread_join(thread_get, NULL);
```

```
    pthread_join(thread_increment, NULL);

    return 1;
}
```

In this example, Bug Finder detects the creation of separate threads with `pthread_create`. The **Data race** defect is raised because the operation `count = count + 1` in the thread with id `thread_increment` conflicts with the operation `c = count` in the thread with id `thread_get`. The variable `count` is accessed in multiple threads without a common protection.

The two conflicting operations are nonatomic. The operation `c = count` is nonatomic on 32-bit targets. See “Define Atomic Operations in Multitasking Code”.

Correction — Protect Operations with `pthread_mutex_lock` and `pthread_mutex_unlock` Pair

To prevent concurrent access on the variable `count`, protect operations on `count` with a critical section. Use the functions `pthread_mutex_lock` and `pthread_mutex_unlock` to implement the critical section.

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;


void* increment_count(void* args)
{
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}

void* set_count(void *args)
{
    long long c;
    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);

    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

Check Information

Group: Rule 14. Concurrency (CON)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

CON43-C

Introduced in R2019a

CERT C: Rule MSC30-C

Do not use the rand() function for generating pseudorandom numbers

Description

Rule Definition

Do not use the rand() function for generating pseudorandom numbers.

Polyspace Implementation

This checker checks for **Vulnerable pseudo-random number generator**.

Examples

Vulnerable pseudo-random number generator

Issue

The **Vulnerable pseudo-random number generator** identifies uses of cryptographically weak pseudo-random number generator (PRNG) routines.

The list of cryptographically weak routines flagged by this checker include:

- `rand`, `random`
- `drand48`, `lrand48`, `mrand48`, `erand48`, `nrand48`, `jrand48`, and their `_r` equivalents such as `drand48_r`
- `RAND_pseudo_bytes`

Risk

These cryptographically weak routines are predictable and must not be used for security purposes. When a predictable random value controls the execution flow, your program is vulnerable to malicious attacks.

Fix

Use more cryptographically sound random number generators, such as `CryptGenRandom` (Windows), `OpenSSL/RAND_bytes`(Linux/UNIX).

Example - Random Loop Numbers

```
#include <stdio.h>
#include <stdlib.h>

volatile int rd = 1;
int main(int argc, char *argv[])
{
    int j, r, nloops;
    struct random_data buf;
    int i = 0;
```

```
    nloops = rand();

    for (j = 0; j < nloops; j++) {
        if (random_r(&buf, &i))
            exit(1);
        printf("random_r: %ld\n", (long)i);
    }
    return 0;
}
```

This example uses `rand` and `random_r` to generate random numbers. If you use these functions for security purposes, these PRNGs can be the source of malicious attacks.

Correction — Use Stronger PRNG

One possible correction is to replace the vulnerable PRNG with a stronger random number generator.

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/rand.h>

volatile int rd = 1;
int main(int argc, char* argv[])
{
    int j, r, nloops;
    unsigned char buf;
    unsigned int seed;
    int i = 0;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <seed> <nloops>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    seed = atoi(argv[1]);
    nloops = atoi(argv[2]);

    for (j = 0; j < nloops; j++) {
        if (RAND_bytes(&buf, i) != 1)
            exit(1);
        printf("RAND_bytes: %u\n", (unsigned)buf);
    }
    return 0;
}
```

Check Information

Group: Rule 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites
MSC30-C

Introduced in R2019a

CERT C: Rule MSC32-C

Properly seed pseudorandom number generators

Description

Rule Definition

Properly seed pseudorandom number generators.

Polyspace Implementation

This checker checks for these issues:

- **Deterministic random output from constant seed**.
- **Predictable random output from predictable seed**.

Examples

Deterministic random output from constant seed

Issue

Deterministic random output from constant seed detects random standard functions that when given a constant seed, have deterministic output.

Risk

When some random functions, such as `srand`, `srandom`, and `initstate`, have constant seeds, the results produce the same output every time that your program is run. A hacker can disrupt your program if they know how your program behaves.

Fix

Use a different random standard function or use a nonconstant seed.

Some standard random routines are inherently cryptographically weak on page 3-405, and should not be used for security purposes.

Example - Random Number Generator Initialization

```
#include <stdlib.h>

void random_num(void)
{
    srand(12345U);
    /* ... */
}
```

This example initializes a random number generator using `srand` with a constant seed. The random number generation is deterministic, making this function cryptographically weak.

Correction — Use Different Random Number Generator

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S
#include <stdlib.h>
#include <stdio.h>

unsigned int random_num_time(void)
{

    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

Predictable random output from predictable seed

Issue

Predictable random output from predictable seed looks for random standard functions that use a nonconstant but predictable seed. Examples of predictable seed generators are `time`, `gettimeofday`, and `getpid`.

Risk

When you use predictable seed values for random number generation, your random numbers are also predictable. A hacker can disrupt your program if they know how your program behaves.

Fix

You can use a different function to generate less predictable seeds.

You can also use a different random number generator that does not require a seed. For example, the Windows API function `rand_s` seeds itself by default. It uses information from the entire system, for example, system time, thread ids, system counter, and memory clusters. This information is more random and a user cannot access this information.

Some standard random routines are inherently cryptographically weak on page 3-405, and should not be used for security purposes.

Example - Seed as an Argument

```
#include <stdlib.h>
#include <time.h>

void seed_rng(int seed)
```

```
{
    srand(seed);
}

int generate_num(void)
{
    seed_rng(time(NULL) + 3);
    /* ... */
}
```

This example uses `srand` to start the random number generator with `seed` as the seed. However, `seed` is predictable because the function `time` generates it. So, an attacker can predict the random numbers generated by `srand`.

Correction — Use Different Random Number Generator

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int generate_num(void)
{
    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

Check Information

Group: Rule 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MSC32-C

Introduced in R2019a

CERT C: Rule MSC33-C

Do not pass invalid data to the asctime() function

Description

Rule Definition

Do not pass invalid data to the asctime() function.

Polyspace Implementation

This checker checks for **Use of obsolete standard function**.

Examples

Use of obsolete standard function

Issue

Use of obsolete standard function detects calls to standard function routines that are considered legacy, removed, deprecated, or obsolete by C/C++ coding standards.

Obsolete Function	Standards	Risk	Replacement Function
`asctime`	Deprecated in POSIX.1-2008	Not thread-safe.	`strftime` or `asctime_s`
`asctime_r`	Deprecated in POSIX.1-2008	Implementation based on unsafe function `sprintf`.	`strftime` or `asctime_s`
`bcmp`	Deprecated in 4.3BSD Marked as legacy in POSIX.1-2001.	Returns from function after finding the first differing byte, making it vulnerable to timing attacks.	`memcmp`
`bcopy`	Deprecated in 4.3BSD Marked as legacy in POSIX.1-2001.	Returns from function after finding the first differing byte, making it vulnerable to timing attacks.	`memcpy` or `memmove`
`brk` and `sbrk`	Marked as legacy in SUSv2 and POSIX.1-2001.		`malloc`
`bsd_signal`	Removed in POSIX.1-2008		`sigaction`
`bzero`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`memset`
`ctime`	Deprecated in POSIX.1-2008	Not thread-safe.	`strftime` or `asctime_s`

Obsolete Function	Standards	Risk	Replacement Function
`ctime_r`	Deprecated in POSIX.1-2008	Implementation based on unsafe function `sprintf`.	`strftime` or `asctime_s`
`cuserid`	Removed in POSIX.1-2001.	Not reentrant. Precise functionality not standardized causing portability issues.	`getpwuid`
`ecvt` and `fcvt`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008	Not reentrant	`snprintf`
`ecvt_r` and `fcvt_r`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008		`snprintf`
`ftime`	Removed in POSIX.1-2008		`time`, `gettimeofday`, `clock_gettime`
`gamma`, `gammaf`, `gammal`	Function not specified in any standard because of historical variations	Portability issues.	`tgamma`, `lgamma`
`gcvt`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`snprintf`
`getcontext`	Removed in POSIX.1-2008.	Portability issues.	Use POSIX thread instead.
`getdtablesize`	BSD API function not included in POSIX.1-2001	Portability issues.	`sysconf( _SC_OPEN _MAX )`
`gethostbyaddr`	Removed in POSIX.1-2008	Not reentrant	`getaddrinfo`
`gethostbyname`	Removed in POSIX.1-2008	Not reentrant	`getnameinfo`
`getpagesize`	BSD API function not included in POSIX.1-2001	Portability issues.	`sysconf( _SC_PAGE SIZE )`
`getpass`	Removed in POSIX.1-2001.	Not reentrant.	`getpwuid`
`getw`	Not present in POSIX.1-2001.		`fread`
`getwd`	Marked legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`getcwd`
`index`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`strchr`
`makecontext`	Removed in POSIX.1-2008.	Portability issues.	Use POSIX thread instead.
`memalign`	Appears in SunOS 4.1.3. Not in 4.4 BSD or POSIX.1-2001		`posix_memalign`
`mktemp`	Removed in POSIX.1-2008.	Generated names are predictable and can cause a race condition.	`mkstemp` removes race risk

Obsolete Function	Standards	Risk	Replacement Function
`pthread_attr_getstackaddr` and `pthread_attr_setstackaddr`		Ambiguities in the specification of the `stackaddr` attribute cause portability issues	`pthread_attr_getstack` and `pthread_attr_setstack`
`putw`	Not present in POSIX.1-2001.	Portability issues.	`fwrite`
`qecvt` and `qfcvt`	Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008		`snprintf`
`qecvt_r` and `qfcvt_r`	Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008		`snprintf`
`rand_r`	Marked as obsolete in POSIX.1-2008		
`re_comp`	BSD API function	Portability issues	`regcomp`
`re_exes`	BSD API function	Portability issues	`regexec`
`rindex`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`strrchr`
`scalb`	Removed in POSIX.1-2008		`scalbln`, `scalblnf`, or `scalblnl`
`sigblock`	4.3BSD signal API whose origin is unclear		`sigprocmask`
`sigmask`	4.3BSD signal API whose origin is unclear		`sigprocmask`
`sigsetmask`	4.3BSD signal API whose origin is unclear		`sigprocmask`
`sigstack`	Interface is obsolete and not implemented on most platforms.	Portability issues.	`sigaltstack`
`sigvec`	4.3BSD signal API whose origin is unclear		`sigaction`
`swapcontext`	Removed in POSIX.1-2008	Portability issues.	Use POSIX threads.
`tmpnam` and `tmpnam_r`	Marked as obsolete in POSIX.1-2008.	This function generates a different string each time it is called, up to TMP_MAX times. If it is called more than TMP_MAX times, the behavior is implementation-defined.	`mkstemp`, `tmpfile`
`ttyslot`	Removed in POSIX.1-2001.		
`ualarm`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.	Errors are under-specified	`setitimer` or POSIX `timer_create`
`usleep`	Removed in POSIX.1-2008.		`nanosleep`
`utime`	SVr4, POSIX.1-2001. POSIX.1-2008 marks as obsolete.		

Obsolete Function	Standards	Risk	Replacement Function
`valloc`	Marked as obsolete in 4.3BSD. Marked as legacy in SUSv2. Removed from POSIX.1-2001		`posix_memalign`
`vfork`	Removed from POSIX.1-2008	Under-specified in previous standards.	`fork`
`wcswcs`	This function was not included in the final ISO/IEC 9899:1990/ Amendment 1:1995 (E).		`wcsstr`
`WinExec`	WinAPI provides this function only for 16-bit Windows compatibility.		`CreateProcess`
`LoadModule`	WinAPI provides this function only for 16-bit Windows compatibility.		`CreateProcess`

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Printing Out Time

```
#include <stdio.h>
#include <time.h>

void timecheck_bad(int argc, char *argv[])
{
    time_t ticks;

    ticks = time(NULL);
    printf("%.24s\r\n", ctime(&ticks));
}
```

In this example, the function `ctime` formats the current time and prints it out. However, `ctime` was removed after C99 because it does not work on multithreaded programs.

Correction — Different Time Function

One possible correction is to use `strftime` instead because this function uses a set buffer size.

```
#include <stdio.h>
#include <string.h>
#include <time.h>

void timecheck_good(int argc, char *argv[])
{
```

```
    char outBuff[1025];
    time_t ticks;
    struct tm * timeinfo;

    memset(outBuff, 0, sizeof(outBuff));

    ticks = time(NULL);
    timeinfo = localtime(&ticks);
    strftime(outBuff,sizeof(outBuff),"%I:%M%p.",timeinfo);
    fprintf(stdout, outBuff);
}
```

Check Information

Group: Rule 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MSC33-C

Introduced in R2019a

CERT C: Rule MSC37-C

Ensure that control never reaches the end of a non-void function

Description

Rule Definition

Ensure that control never reaches the end of a non-void function.

Polyspace Implementation

This checker checks for **Missing return statement**.

Examples

Missing return statement

Issue

Missing return statement occurs when a function does not return a value along at least one execution path. If the return type of the function is `void`, this error does not occur.

Risk

If a function has a non-`void` return value in its signature, it is expected to return a value. The return value of this function can be used in later computations. If the execution of the function body goes through a path where a `return` statement is missing, the function return value is indeterminate. Computations with this return value can lead to unpredictable results.

Fix

In most cases, you can fix this defect by placing the `return` statement at the end of the function body.

Alternatively, you can identify which execution paths through the function body do not have a `return` statement and add a `return` statement on those paths. Often the result details show a sequence of events that indicate this execution path. You can add a `return` statement at an appropriate point in the path. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Missing or invalid return statement error

```
int AddSquares(int n)
 {
   int i=0;
   int sum=0;
```

```
   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }
 }
/* Defect: No return value if n is not 0*/
```

If `n` is equal to 0, the code does not enter the `if` statement. Therefore, the function `AddSquares` does not return a value if `n` is 0.

Correction — Place Return Statement on Every Execution Path

One possible correction is to return a value in every branch of the `if...else` statement.

```
 int AddSquares(int n)
 {
   int i=0;
   int sum=0;

   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }

   /*Fix: Place a return statement on branches of if-else */
   else
     return 0;
  }
```

Check Information

Group: Rule 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MSC37-C

Introduced in R2019a

CERT C: Rule MSC38-C

Do not treat a predefined identifier as an object if it might only be implemented as a macro

Description

Rule Definition

Do not treat a predefined identifier as an object if it might only be implemented as a macro.

Polyspace Implementation

This checker checks for **Predefined macro used as an object**.

Examples

Predefined macro used as an object

Issue

Predefined macro used as an object occurs when you use certain identifiers in a way that requires an underlying object to be present. These identifiers are defined as macros. The C Standard does not allow you to redefine them as objects. You use the identifiers in such a way that macro expansion of the identifiers cannot occur.

For instance, you refer to an external variable `errno`:

```
extern int errno;
```

However, `errno` does not occur as a variable but a macro.

The defect applies to these macros: `assert`, `errno`, `math_errhandling`, `setjmp`, `va_arg`, `va_copy`, `va_end`, and `va_start`. The checker looks for the defect only in source files (not header files).

Risk

The C11 Standard (Sec. 7.1.4) allows you to redefine most macros as objects. To access the object and not the macro in a source file, you do one of these:

- Redeclare the identifier as an external variable or function.
- For function-like macros, enclose the identifier name in parentheses.

If you try to use these strategies for macros that cannot be redefined as objects, an error occurs.

Fix

Do not use the identifiers in such a way that a macro expansion is suppressed.

- Do not redeclare the identifiers as external variables or functions.
- For function-like macros, do not enclose the macro name in parentheses.

Example - Use of `assert` as Function

```
#include<assert.h>
typedef void (*err_handler_func)(int);

extern void demo_handle_err(err_handler_func, int);

void func(int err_code) {
    extern void assert(int);
    demo_handle_err(&(assert), err_code);
}
```

In this example, the `assert` macro is redefined as an external function. When passed as an argument to `demo_handle_err`, the identifier `assert` is enclosed in parentheses, which suppresses use of the `assert` macro.

Correction — Use `assert` as Macro

One possible correction is to directly use the `assert` macro from `assert.h`. A different implementation of the function `demo_handle_err` directly uses the `assert` macro instead of taking the address of an `assert` function.

```
#include<assert.h>
void demo_handle_err(int err_code) {
    assert(err_code == 0);
}

void func(int err_code) {
    demo_handle_err(err_code);
}
```

Check Information

Group: Rule 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MSC38-C

Introduced in R2019a

CERT C: Rule MSC39-C

Do not call va_arg() on a va_list that has an indeterminate value

Description

Rule Definition

Do not call va_arg() on a va_list that has an indeterminate value.

Polyspace Implementation

This checker checks for these issues:

- **Invalid va_list argument**.
- **Too many va_arg calls for current argument list**.

Examples

Invalid va_list argument

Issue

Invalid va_list argument occurs when you use a `va_list` variable as an argument to a function in the `vprintf` group but:

- You do not initialize the variable previously using `va_start` or `va_copy`.
- You invalidate the variable previously using `va_end` and do not reinitialize it.

For instance, you call the function `vsprintf` as `vsprintf (buffer,format, args)`. However, before the function call, you do not initialize the `va_list` variable `args` using either of the following:

- `va_start(args, paramName)`. `paramName` is the last named argument of a variable-argument function. For instance, for the function definition `void func(int n, char c, ...) {}`, `c` is the last named argument.
- `va_copy(args, anotherList)`. `anotherList` is another valid `va_list` variable.

Risk

The behavior of an uninitialized `va_list` argument is undefined. Calling a function with an uninitialized `va_list` argument can cause stack overflows.

Fix

Before using a `va_list` variable as function argument, initialize it with `va_start` or `va_copy`.

Clean up the variable using `va_end` only after all uses of the variable.

Example - `va_list` Variable Used Following Call to `va_end`

```
#include <stdarg.h>
#include <stdio.h>
```

```
int call_vfprintf(int line, const char *format, ...) {
    va_list ap;
    int r=0;

    va_start(ap, format);
    r = vfprintf(stderr, format, ap);
    va_end(ap);

    r += vfprintf(stderr, format, ap);
    return r;
}
```

In this example, the `va_list` variable `ap` is used in the `vfprintf` function, after the `va_end` macro is called.

Correction — Call `va_end` After Using `va_list` Variable

One possible correction is to call `va_end` only after all uses of the `va_list` variable.

```
#include <stdarg.h>
#include <stdio.h>

int call_vfprintf(int line, const char *format, ...) {
    va_list ap;
    int r=0;

    va_start(ap, format);
    r = vfprintf(stderr, format, ap);
    r += vfprintf(stderr, format, ap);
    va_end(ap);

    return r;
}
```

Too many va_arg calls for current argument list

Issue

Too many va_arg calls for current argument list occurs when the number of calls to `va_arg` exceeds the number of arguments passed to the corresponding variadic function. The analysis raises a defect only when the variadic function is called.

Too many va_arg calls for current argument list does not raise a defect when:

- The number of calls to `va_arg` inside the variadic function is indeterminate. For example, if the calls are from an external source.
- The `va_list` used in `va_arg` is invalid.

Risk

When you call `va_arg` and there is no next argument available in `va_list`, the behavior is undefined. The call to `va_arg` might corrupt data or return an unexpected result.

Fix

Ensure that you pass the correct number of arguments to the variadic function.

Example - No Argument Available When Calling `va_arg`

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/
int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
        if (count > 0) {
/* No further argument available
* in va_list when calling va_arg
*/

            result += va_arg(ap, int);
        }
    }
    va_end(ap);
    return result;
}

void func(void) {

    (void)variadic_func(2, 100);

}
```

In this example, the named argument and only one variadic argument are passed to `variadic_func()` when it is called inside `func()`. On the second call to `va_arg`, no further variadic argument is available in `ap` and the behavior is undefined.

Correction — Pass Correct Number of Arguments to Variadic Function

One possible correction is to ensure that you pass the correct number of arguments to the variadic function.

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/

int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
```

```
        if (count > 0) {

/* The correct number of arguments is
* passed to va_list when variadic_func()
* is called inside func()
*/
            result += va_arg(ap, int);
        }
    }
    va_end(ap);
    return result;
}

void func(void) {

    (void)variadic_func(2, 100, 200);

}
```

Check Information

Group: Rule 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MSC39-C

Introduced in R2019a

CERT C: Rule MSC40-C

Do not violate constraints

Description

Rule Definition

Do not violate constraints.

Polyspace Implementation

This checker checks for **Inline constraint not respected**.

Examples

Inline constraint not respected

Issue

Inline constraint not respected occurs when you refer to a file scope modifiable static variable or define a local modifiable static variable in a nonstatic inlined function. The checker considers a variable as modifiable if it is not `const`-qualified.

For instance, `var` is a modifiable `static` variable defined in an `inline` function `func`. `g_step` is a file scope modifiable static variable referred to in the same inlined function.

```
static int g_step;
inline void func (void) {
   static int var = 0;
   var += g_step;
}
```

Risk

When you modify a static variable in multiple function calls, you expect to modify the same variable in each call. For instance, each time you call `func`, the same instance of `var1` is incremented but a separate instance of `var2` is incremented.

```
void func(void) {
   static var1 = 0;
   var2 = 0;
   var1++;
   var2++;
}
```

If a function has an inlined and non-inlined definition (in separate files), when you call the function, the C standard allows compilers to use either the inlined or the non-inlined form (see ISO/IEC 9899:2011, sec. 6.7.4). If your compiler uses an inlined definition in one call and the non-inlined definition in another, you are no longer modifying the same variable in both calls. This behavior defies the expectations from a static variable.

Fix

Use one of these fixes:

- If you do not intend to modify the variable, declare it as `const`.

  If you do not modify the variable, there is no question of unexpected modification.
- Make the variable non-`static`. Remove the `static` qualifier from the declaration.

  If the variable is defined in the function, it becomes a regular local variable. If defined at file scope, it becomes an extern variable. Make sure that this change in behavior is what you intend.
- Make the function `static`. Add a `static` qualifier to the function definition.

  If you make the function `static`, the file with the inlined definition always uses the inlined definition when the function is called. Other files use another definition of the function. The question of which function definition gets used is not left to the compiler.

Example - Static Variable Use in Inlined and External Definition

```
/* file1. c  : contains inline definition of get_random()*/

inline unsigned int get_random(void)
{

    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}


int call_get_random(void)
{
    unsigned int rand_no;
    int ii;
    for (ii = 0; ii < 100; ii++) {
         rand_no = get_random();
    }
    rand_no = get_random();
    return 0;
}

/* file2. c  : contains external definition of get_random()*/

extern unsigned int get_random(void)
{
    /* Initialize seeds */
    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}
```

In this example, `get_random()` has an inline definition in `file1.c` and an external definition in `file2.c`. When `get_random` is called in `file1.c`, compilers are free to choose whether to use the inline or the external definition.

Depending on the definition used, you might or might not modify the version of `m_z` and `m_w` in the inlined version of `get_random()`. This behavior contradicts the usual expectations from a static variable. When you call `get_random()`, you expect to always modify the same `m_z` and `m_w`.

Correction — Make Inlined Function Static

One possible correction is to make the inlined `get_random()` static. Irrespective of your compiler, calls to `get_random()` in `file1.c` then use the inlined definition. Calls to `get_random()` in other files use the external definition. This fix removes the ambiguity about which definition is used and whether the static variables in that definition are modified.

```
/* file1. c  : contains inline definition of get_random()*/

static inline unsigned int get_random(void)
{

    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}


int call_get_random(void)
{
    unsigned int rand_no;
    int ii;
    for (ii = 0; ii < 100; ii++) {
         rand_no = get_random();
    }
    rand_no = get_random();
    return 0;
}

/* file2. c  : contains external definition of get_random()*/

extern unsigned int get_random(void)
{
    /* Initialize seeds */
    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}
```

Check Information

Group: Rule 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MSC40-C

Introduced in R2019a

CERT C: Rule MSC41-C

Never hard code sensitive information

Description

Rule Definition

Never hard code sensitive information.

Polyspace Implementation

This checker checks for **Hard coded sensitive data**.

Examples

Hard coded sensitive data

Hard coded sensitive data occurs when data that is potentially sensitive is directly exposed in the code, for instance, as string literals. The checker identifies data as sensitive from their use in certain functions such as password encryption functions.

Following data can be potentially sensitive.

Type of Data	Functions That Indicate Sensitive Nature of Information
Host name	• `sethostname`, `setdomainname`, `gethostbyname`, `gethostbyname2`, `getaddrinfo`, `gethostbyname_r`, `gethostbyname2_r` (string argument) • `inet_aton`, `inet_pton`, `inet_net_pton`, `inet_addr`, `inet_network` (string argument) • `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (2nd argument)
Password	• `CreateProcessWithLogonW`, `LogonUser` (1st argument) • `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (3rd argument)

Type of Data	Functions That Indicate Sensitive Nature of Information
Database	• MySQL: `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (4th argument) • SQLite: `sqlite3_open`, `sqlite3_open16`, `sqlite3_open_v2` (1st argument) • PostgreSQL: `PQconnectdb` • Microsoft SQL: `SQLDriverConnect` (3rd argument)
User name	• `getpw`, `getpwnam`, `getpwnam_r`, `getpwuid`, `getpwuid_r`
Salt	`crypt`, `crypt_r` (2nd argument)
Cryptography keys and initialization vectors	OpenSSL: • `EVP_CipherInit`, `EVP_EncryptInit`, `EVP_DecryptInit` (3rd argument) • `EVP_CipherInit_ex`, `EVP_EncryptInit_ex`, `EVP_DecryptInit_ex` (4th argument)
Seed	• `srand`, `srandom`, `initstate` (1st argument) • OpenSSL: `RAND_seed`, `RAND_add`

Risk

Information that is hardcoded can be queried from binaries generated from the code.

Fix

Avoid hard coding sensitive information.

Example – Sensitive Data Exposed Through String Literals

```
// Typically, you include the header "mysql.h" with function and type declarations.
// In this example, only the required lines from the header are quoted.

typedef struct _MYSQL MYSQL;

MYSQL *mysql_real_connect(MYSQL *mysql,
                          const char *host, const char *user, const char *passwd,
                          const char *db, unsigned int port, const char *unix_socket,
                          unsigned long client_flag);

typedef void * DbHandle;
extern MYSQL *sql;

// File that uses functions from "mysql.h"
const char *host = "localhost";
char *user = "guest";
char *passwd;

DbHandle connect_to_database_server(const char *db)
```

```
{
    passwd = (char*)"guest";
    return (DbHandle)
        mysql_real_connect (sql, host, user, passwd, db, 0, 0x0, 0);
}
```

In this example, the `mysql_real_connect` arguments `host` (host name), `user` (user name), and `passwd` (password) are string literals and directly exposed in the code.

Querying the generated binary for ASCII strings can reveal this information.

Correction – Read Sensitive Data from Secured Configuration Files

One possible correction is to read the data from a configuration file. In the following corrected example, the call to function `connect_to_database_server_init` presumably reads the host name, user name, and password into its arguments from a secured configuration file.

```
// Typically, you include the header "mysql.h" with function and type declarations.
// In this example, only the required lines from the header are quoted.

typedef struct _MYSQL MYSQL;

MYSQL *mysql_real_connect(MYSQL *mysql,
                          const char *host, const char *user, const char *passwd,
                          const char *db, unsigned int port, const char *unix_socket,
                          unsigned long client_flag);

typedef void * DbHandle;
extern MYSQL *sql;

// File that uses functions from "mysql.h"

DbHandle connect_to_database_server(const char *db)
{
    const char *host_from_cfg;
    const char *user_from_cfg;
    const char *passwd_from_cfg;
    const char *db_from_cfg;
    if (connect_to_database_server_init(&host_from_cfg,
                                        &user_from_cfg,
                                        &passwd_from_cfg,
                                        &db_from_cfg))
    {
        return (DbHandle)
            mysql_real_connect (sql, host_from_cfg, user_from_cfg,
                            passwd_from_cfg, db_from_cfg,  0, 0x0, 0);
    }
    else
        return (DbHandle)0x0;
}
```

Check Information

Group: Rule 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics
“Check for Coding Standard Violations”

External Websites
MSC41-C

Introduced in R2020a

CERT C: Rule POS30-C

Use the readlink() function properly

Description

Rule Definition

Use the readlink() function properly.

Polyspace Implementation

This checker checks for **Misuse of readlink()**.

Examples

Misuse of readlink()

Issue

Misuse of readlink() occurs when you pass a buffer size argument to `readlink()` that does not leave space for a null terminator in the buffer.

For instance:

```
ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf));
```

The third argument is exactly equal to the size of the second argument. For large enough symbolic links, this use of `readlink()` does not leave space to enter a null terminator.

Risk

The `readlink()` function copies the content of a symbolic link (first argument) to a buffer (second argument). However, the function does not append a null terminator to the copied content. After using `readlink()`, you must explicitly add a null terminator to the buffer.

If you fill the entire buffer when using `readlink`, you do not leave space for this null terminator.

Fix

When using the `readlink()` function, make sure that the third argument is one less than the buffer size.

Then, append a null terminator to the buffer. To determine where to add the null terminator, check the return value of `readlink()`. If the return value is -1, an error has occurred. Otherwise, the return value is the number of characters (bytes) copied.

Example - Incorrect Size Argument of `readlink`

```
#include <unistd.h>

#define SIZE1024 1024

extern void display_path(const char *);
```

```
void func() {
    char buf[SIZE1024];
    ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf));
    if (len > 0) {
        buf[len - 1] = '\0';
    }
    display_path(buf);
}
```

In this example, the third argument of `readlink` is exactly the size of the buffer (second argument). If the first argument is long enough, this use of `readlink` does not leave space for the null terminator.

Also, if no characters are copied, the return value of `readlink` is 0. The following statement leads to a buffer underflow when `len` is 0.

```
buf[len - 1] = '\0';
```

Correction — Make Sure Size Argument is One Less Than Buffer Size

One possible correction is to make sure that the third argument of `readlink` is one less than size of the second argument.

The following corrected code also accounts for `readlink` returning 0.

```
#include <stdlib.h>
#include <unistd.h>

#define fatal_error() abort()
#define SIZE1024 1024

extern void display_path(const char *);

void func() {
    char buf[SIZE1024];
    ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf) - 1);
    if (len != -1) {
        buf[len] = '\0';
        display_path(buf);
    }
    else {
        /* Handle error */
        fatal_error();
    }
}
```

Check Information

Group: Rule 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites
POS30-C

Introduced in R2019a

CERT C: Rule POS33-C

Do not use vfork()

Description

Rule Definition

Do not use vfork().

Polyspace Implementation

This checker checks for **Use of obsolete standard function**.

Examples

Use of obsolete standard function

Issue

Use of obsolete standard function detects calls to standard function routines that are considered legacy, removed, deprecated, or obsolete by C/C++ coding standards.

Obsolete Function	Standards	Risk	Replacement Function
`asctime`	Deprecated in POSIX.1-2008	Not thread-safe.	`strftime` or `asctime_s`
`asctime_r`	Deprecated in POSIX.1-2008	Implementation based on unsafe function `sprintf`.	`strftime` or `asctime_s`
`bcmp`	Deprecated in 4.3BSD Marked as legacy in POSIX.1-2001.	Returns from function after finding the first differing byte, making it vulnerable to timing attacks.	`memcmp`
`bcopy`	Deprecated in 4.3BSD Marked as legacy in POSIX.1-2001.	Returns from function after finding the first differing byte, making it vulnerable to timing attacks.	`memcpy` or `memmove`
`brk` and `sbrk`	Marked as legacy in SUSv2 and POSIX.1-2001.		`malloc`
`bsd_signal`	Removed in POSIX.1-2008		`sigaction`
`bzero`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`memset`
`ctime`	Deprecated in POSIX.1-2008	Not thread-safe.	`strftime` or `asctime_s`

Obsolete Function	Standards	Risk	Replacement Function
`ctime_r`	Deprecated in POSIX.1-2008	Implementation based on unsafe function `sprintf`.	`strftime` or `asctime_s`
`cuserid`	Removed in POSIX.1-2001.	Not reentrant. Precise functionality not standardized causing portability issues.	`getpwuid`
`ecvt` and `fcvt`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008	Not reentrant	`snprintf`
`ecvt_r` and `fcvt_r`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008		`snprintf`
`ftime`	Removed in POSIX.1-2008		`time`, `gettimeofday`, `clock_gettime`
`gamma`, `gammaf`, `gammal`	Function not specified in any standard because of historical variations	Portability issues.	`tgamma`, `lgamma`
`gcvt`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`snprintf`
`getcontext`	Removed in POSIX.1-2008.	Portability issues.	Use POSIX thread instead.
`getdtablesize`	BSD API function not included in POSIX.1-2001	Portability issues.	`sysconf( _SC_OPEN_MAX )`
`gethostbyaddr`	Removed in POSIX.1-2008	Not reentrant	`getaddrinfo`
`gethostbyname`	Removed in POSIX.1-2008	Not reentrant	`getnameinfo`
`getpagesize`	BSD API function not included in POSIX.1-2001	Portability issues.	`sysconf( _SC_PAGESIZE )`
`getpass`	Removed in POSIX.1-2001.	Not reentrant.	`getpwuid`
`getw`	Not present in POSIX.1-2001.		`fread`
`getwd`	Marked legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`getcwd`
`index`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`strchr`
`makecontext`	Removed in POSIX.1-2008.	Portability issues.	Use POSIX thread instead.
`memalign`	Appears in SunOS 4.1.3. Not in 4.4 BSD or POSIX.1-2001		`posix_memalign`
`mktemp`	Removed in POSIX.1-2008.	Generated names are predictable and can cause a race condition.	`mkstemp` removes race risk

Obsolete Function	Standards	Risk	Replacement Function
`pthread_attr_getstackaddr` and `pthread_attr_setstackaddr`		Ambiguities in the specification of the `stackaddr` attribute cause portability issues	`pthread_attr_getstack` and `pthread_attr_setstack`
`putw`	Not present in POSIX.1-2001.	Portability issues.	`fwrite`
`qecvt` and `qfcvt`	Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008		`snprintf`
`qecvt_r` and `qfcvt_r`	Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008		`snprintf`
`rand_r`	Marked as obsolete in POSIX.1-2008		
`re_comp`	BSD API function	Portability issues	`regcomp`
`re_exes`	BSD API function	Portability issues	`regexec`
`rindex`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`strrchr`
`scalb`	Removed in POSIX.1-2008		`scalbln`, `scalblnf`, or `scalblnl`
`sigblock`	4.3BSD signal API whose origin is unclear		`sigprocmask`
`sigmask`	4.3BSD signal API whose origin is unclear		`sigprocmask`
`sigsetmask`	4.3BSD signal API whose origin is unclear		`sigprocmask`
`sigstack`	Interface is obsolete and not implemented on most platforms.	Portability issues.	`sigaltstack`
`sigvec`	4.3BSD signal API whose origin is unclear		`sigaction`
`swapcontext`	Removed in POSIX.1-2008	Portability issues.	Use POSIX threads.
`tmpnam` and `tmpnam_r`	Marked as obsolete in POSIX.1-2008.	This function generates a different string each time it is called, up to TMP_MAX times. If it is called more than TMP_MAX times, the behavior is implementation-defined.	`mkstemp`, `tmpfile`
`ttyslot`	Removed in POSIX.1-2001.		
`ualarm`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.	Errors are under-specified	`setitimer` or POSIX `timer_create`
`usleep`	Removed in POSIX.1-2008.		`nanosleep`
`utime`	SVr4, POSIX.1-2001. POSIX.1-2008 marks as obsolete.		

Obsolete Function	Standards	Risk	Replacement Function
`valloc`	Marked as obsolete in 4.3BSD. Marked as legacy in SUSv2. Removed from POSIX.1-2001		`posix_memalign`
`vfork`	Removed from POSIX.1-2008	Under-specified in previous standards.	`fork`
`wcswcs`	This function was not included in the final ISO/IEC 9899:1990/ Amendment 1:1995 (E).		`wcsstr`
`WinExec`	WinAPI provides this function only for 16-bit Windows compatibility.		`CreateProcess`
`LoadModule`	WinAPI provides this function only for 16-bit Windows compatibility.		`CreateProcess`

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Printing Out Time

```
#include <stdio.h>
#include <time.h>

void timecheck_bad(int argc, char *argv[])
{
    time_t ticks;

    ticks = time(NULL);
    printf("%.24s\r\n", ctime(&ticks));
}
```

In this example, the function `ctime` formats the current time and prints it out. However, `ctime` was removed after C99 because it does not work on multithreaded programs.

Correction — Different Time Function

One possible correction is to use `strftime` instead because this function uses a set buffer size.

```
#include <stdio.h>
#include <string.h>
#include <time.h>

void timecheck_good(int argc, char *argv[])
{
```

```
    char outBuff[1025];
    time_t ticks;
    struct tm * timeinfo;

    memset(outBuff, 0, sizeof(outBuff));

    ticks = time(NULL);
    timeinfo = localtime(&ticks);
    strftime(outBuff,sizeof(outBuff),"%I:%M%p.",timeinfo);
    fprintf(stdout, outBuff);
}
```

Check Information

Group: Rule 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

POS33-C

Introduced in R2019a

CERT C: Rule POS34-C

Do not call putenv() with a pointer to an automatic variable as the argument

Description

Rule Definition

Do not call putenv() with a pointer to an automatic variable as the argument.

Polyspace Implementation

This checker checks for **Use of automatic variable as putenv-family function argument**.

Examples

Use of automatic variable as putenv-family function argument

Issue

Use of automatic variable as `putenv`-family function argument occurs when the argument of a `putenv`-family function is a local variable with automatic duration.

Risk

The function `putenv(char *string)` inserts a pointer to its supplied argument into the environment array, instead of making a copy of the argument. If the argument is an automatic variable, its memory can be overwritten after the function containing the `putenv()` call returns. A subsequent call to `getenv()` from another function returns the address of an out-of-scope variable that cannot be dereferenced legally. This out-of-scope variable can cause environment variables to take on unexpected values, cause the program to stop responding, or allow arbitrary code execution vulnerabilities.

Fix

Use `setenv()`/`unsetenv()` to set and unset environment variables. Alternatively, use `putenv`-family function arguments with dynamically allocated memory, or, if your application has no reentrancy requirements, arguments with static duration. For example, a single thread execution with no recursion or interrupts does not require reentrancy. It cannot be called (reentered) during its execution.

Example - Automatic Variable as Argument of `putenv()`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024

void func(int var)
{
    char env[SIZE1024];
    int retval = sprintf(env, "TEST=%s", var ? "1" : "0");
    if (retval <= 0) {
```

```
        /* Handle error */
    }
    /* Environment variable TEST is set using putenv().
    The argument passed to putenv is an automatic variable. */
    retval = putenv(env);
    if (retval != 0) {
        /* Handle error */
    }
}
```

In this example, `sprintf()` stores the character string `TEST=var` in `env`. The value of the environment variable `TEST` is then set to `var` by using `putenv()`. Because `env` is an automatic variable, the value of `TEST` can change once `func()` returns.

Correction — Use `static` Variable for Argument of `putenv()`

Declare `env` as a static-duration variable. The memory location of `env` is not overwritten for the duration of the program, even after `func()` returns.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024
void func(int var)
{
    /* static duration variable */
    static char env[SIZE1024];
    int retval = sprintf(env,"TEST=%s", var ? "1" : "0");
    if (retval <= 0) {
        /* Handle error */
    }

    /* Environment variable TEST is set using putenv() */
    retval=putenv(env);
    if (retval != 0) {
        /* Handle error */
    }
}
```

Correction — Use `setenv()` to Set Environment Variable Value

To set the value of `TEST` to `var`, use `setenv()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024

void func(int var)
{
    /* Environment variable TEST is set using setenv() */
    int retval = setenv("TEST", var ? "1" : "0", 1);

    if (retval != 0) {
        /* Handle error */
```

```
    }
}
```

Check Information

Group: Rule 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

POS34-C

Introduced in R2019a

CERT C: Rule POS35-C

Avoid race conditions while checking for the existence of a symbolic link

Description

Rule Definition

Avoid race conditions while checking for the existence of a symbolic link.

Polyspace Implementation

This checker checks for **File access between time of check and use (TOCTOU)**.

Examples

File access between time of check and use (TOCTOU)

Issue

File access between time of check and use (TOCTOU) detects race condition issues between checking the existence of a file or folder, and using a file or folder.

Risk

An attacker can access and manipulate your file between your check for the file and your use of a file. Symbolic links are particularly risky because an attacker can change where your symbolic link points.

Fix

Before using a file, do not check its status. Instead, use the file and check the results afterward.

Example - Check File Before Using

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern void print_tofile(FILE* f);

void toctou(char * log_path) {
    if (access(log_path, W_OK)==0) {
        FILE* f = fopen(log_path, "w");
        if (f) {
            print_tofile(f);
            fclose(f);
        }
    }
}
```

In this example, before opening and using the file, the function checks if the file exists. However, an attacker can change the file between the first and second lines of the function.

Correction — Open Then Check

One possible correction is to open the file, and then check the existence and contents afterward.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern void print_tofile(FILE* f);

void toctou(char * log_path) {
    int fd = open(log_path, O_WRONLY);
    if (fd!=-1) {
        FILE *f = fdopen(fd, "w");
        if (f) {
            print_tofile(f);
            fclose(f);
        }
    }
}
```

Check Information

Group: Rule 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

POS35-C

Introduced in R2019a

CERT C: Rule POS36-C

Observe correct revocation order while relinquishing privileges

Description

Rule Definition

Observe correct revocation order while relinquishing privileges.

Polyspace Implementation

This checker checks for **Bad order of dropping privileges**.

Examples

Bad order of dropping privileges

Issue

Bad order of dropping privileges checks the order of privilege drops. If you drop higher elevated privileges before dropping lower elevated privileges, Polyspace raises a defect. For example dropping elevated primary group privileges before dropping elevated ancillary group privileges.

Risk

If you drop privileges in the wrong order, you can potentially drop higher privileges that you need to drop lower privileges. The incorrect order can mean, privileges are not dropped, compromising the security of your program.

Fix

Respect this order of dropping elevated privileges:

- Drop (elevated) ancillary group privileges, then drop (elevated) primary group privileges.
- Drop (elevated) primary group privileges, then drop (elevated) user privileges.

Example - Dropping User Privileges First

```
#define _BSD_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>
#define fatal_error() abort()

static void sanitize_privilege_drop_check(uid_t olduid, gid_t oldgid)
{
    if (seteuid(olduid) != -1)
    {
        /* Privileges can be restored, handle error */
        fatal_error();
    }
    if (setegid(oldgid) != -1)
```

```
    {
        /* Privileges can be restored, handle error */
        fatal_error();
    }
}
void badprivilegedroporder(void) {
    uid_t
        newuid = getuid(),
        olduid = geteuid();
    gid_t
        newgid = getgid(),
        oldgid = getegid();

    if (setuid(newuid) == -1) {
        /* handle error condition */
        fatal_error();
    }
    if (setgid(newgid) == -1)  {
        /* handle error condition */
        fatal_error();
    }
    if (olduid == 0) {
        /* drop ancillary groups IDs only possible for root */
        if (setgroups(1, &newgid) == -1) {
            /* handle error condition */
            fatal_error();
        }
    }

    sanitize_privilege_drop_check(olduid, oldgid);
}
```

In this example, there are two privilege drops made in the incorrect order. `setgid` attempts to drop group privileges. However, `setgid` requires the user privileges, which were dropped previously using `setuid`, to perform this function. After dropping group privileges, this function attempts to drop ancillary groups privileges by using `setgroups`. This task requires the higher primary group privileges that were dropped with `setgid`. At the end of this function, it is possible to regain group privileges because the order of dropping privileges was incorrect.

Correction — Reverse Privilege Drop Order

One possible correction is to drop the lowest level privileges first. In this correction, ancillary group privileges are dropped, then primary group privileges are dropped, and finally user privileges are dropped.

```
#define _BSD_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>
#define fatal_error() abort()

static void sanitize_privilege_drop_check(uid_t olduid, gid_t oldgid)
{
    if (seteuid(olduid) != -1)
    {
        /* Privileges can be restored, handle error */
        fatal_error();
```

```
    }
    if (setegid(oldgid) != -1)
    {
        /* Privileges can be restored, handle error */
        fatal_error();
    }
}
void badprivilegedroporder(void) {
    uid_t
        newuid = getuid(),
        olduid = geteuid();
    gid_t
        newgid = getgid(),
        oldgid = getegid();

    if (olduid == 0) {
        /* drop ancillary groups IDs only possible for root */
        if (setgroups(1, &newgid) == -1) {
            /* handle error condition */
            fatal_error();
        }
    }
    if (setgid(getgid()) == -1)  {
        /* handle error condition */
        fatal_error();
    }
    if (setuid(getuid()) == -1) {
        /* handle error condition */
        fatal_error();
    }

    sanitize_privilege_drop_check(olduid, oldgid);
}
```

Check Information

Group: Rule 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

POS36-C

Introduced in R2019a

CERT C: Rule POS37-C

Ensure that privilege relinquishment is successful

Description

Rule Definition

Ensure that privilege relinquishment is successful.

Polyspace Implementation

This checker checks for **Privilege drop not verified**.

Examples

Privilege drop not verified

Issue

Privilege drop not verified detects calls to functions that relinquish privileges. If you do not verify that the privileges were dropped before the end of your function, a defect is raised.

Risk

If privilege relinquishment fails, an attacker can regain elevated privileges and have more access to your program than intended. This security hole can cause unexpected behavior in your code if left open.

Fix

Before the end of scope, verify that the privileges that you dropped were actually dropped.

Example - Drop Privileges Within a Function

```
#define _BSD_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>
#define fatal_error() abort()
extern int need_more_privileges;

void missingprivilegedropcheck()
{
    /* Code intended to run with elevated privileges */

    /* Temporarily drop elevated privileges */
    if (seteuid(getuid()) != 0) {
        /* Handle error */
        fatal_error();
    }

    /* Code intended to run with lower privileges */
```

```
    if (need_more_privileges) {
        /* Restore elevated privileges */
        if (seteuid(0) != 0) {
            /* Handle error */
            fatal_error();
        }
        /* Code intended to run with elevated privileges */
    }

    /* ... */

    /* Permanently drop elevated privileges */
    if (setuid(getuid()) != 0) {
        /* Handle error */
        fatal_error();
    }

    /* Code intended to run with lower privileges */
}
```

In this example, privileges are elevated and dropped to run code with the intended privilege level. When privileges are dropped, the privilege level before exiting the function body is not verified. A malicious attacker can regain their elevated privileges.

Correction — Verify Privilege Drop

One possible correction is to use `setuid` to verify that the privileges were dropped.

```
#define _BSD_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>
#define fatal_error() abort()
extern int need_more_privileges;

void missingprivilegedropcheck()
{
    /* Store the privileged ID for later verification */
    uid_t privid = geteuid();

    /* Code intended to run with elevated privileges   */

    /* Temporarily drop elevated privileges */
    if (seteuid(getuid()) != 0) {
        /* Handle error */
        fatal_error();
    }

    /* Code intended to run with lower privileges  */

    if (need_more_privileges) {
        /* Restore elevated Privileges */
        if (seteuid(privid) != 0) {
            /* Handle error */
            fatal_error();
        }
        /* Code intended to run with elevated privileges   */
```

```
    }

    /* ... */

    /* Restore privileges if needed */
    if (geteuid() != privid) {
        if (seteuid(privid) != 0) {
            /* Handle error */
            fatal_error();
        }
    }

    /* Permanently drop privileges */
    if (setuid(getuid()) != 0) {
        /* Handle error */
        fatal_error();
    }

    if (setuid(0) != -1) {
        /* Privileges can be restored, handle error */
        fatal_error();
    }

    /* Code intended to run with lower privileges; */
}
```

Check Information

Group: Rule 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

POS37-C

Introduced in R2019a

CERT C: Rule POS38-C

Beware of race conditions when using fork and file descriptors

Description

Rule Definition

Beware of race conditions when using fork and file descriptors.

Polyspace Implementation

This checker checks for **File descriptor exposure to child process**.

Examples

File descriptor exposure to child process

Issue

File descriptor exposure to child process occurs when a process is forked and the child process uses file descriptors inherited from the parent process.

Risk

When you fork a child process, file descriptors are copied from the parent process, which means that you can have concurrent operations on the same file. Use of the same file descriptor in the parent and child processes can lead to race conditions that may not be caught during standard debugging. If you do not properly manage the file descriptor permissions and privileges, the file content is vulnerable to attacks targeting the child process.

Fix

Check that the file has not been modified before forking the process. Close all inherited file descriptors and reopen them with stricter permissions and privileges, such as read-only permission.

Example - File Descriptor Accessed from Forked Process

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <unistd.h>
# include <fcntl.h>
# include <sys/types.h>
# include <sys/stat.h>



const char *test_file="/home/user/test.txt";

void func(void)
{
    char c;
    pid_t pid;
    /* create file descriptor in read and write mode */
```

```
    int fd = open(test_file, O_RDWR);
    if (fd == -1)
    {
        /* Handle error */
        abort();
    }
    /* fork process */
    pid = fork();
    if (pid == -1)
    {
        /* Handle error */
        abort();
    }
    else if (pid == 0)
    {   /* Child process accesses file descriptor inherited
        from parent process */
        (void)read(fd, &c, 1);
    }
    else
    {   /* Parent process access same file descriptor as
        child process */
        (void)read(fd, &c, 1);
    }
}
```

In this example, a file descriptor `fd` is created in read and write mode. The process is then forked. The child process inherits and accesses `fd` with the same permissions as the parent process. A race condition exists between the parent and child processes. The contents of the file is vulnerable to attacks through the child process.

Correction — Close and Reopen Inherited File Descriptor

After you create the file descriptor, check the file for tampering. Then, close the inherited file descriptor in the child process and reopen it in read-only mode.

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <unistd.h>
# include <fcntl.h>
# include <sys/types.h>
# include <sys/stat.h>


const char *test_file="/home/user/test.txt";

void func(void)
{
    char c;
    pid_t pid;

    /* Get the state of file for further file tampering checking */

    /* create file descriptor in read and write mode */
    int fd = open(test_file, O_RDWR);
    if (fd == -1)
    {
```

```
        /* Handle error */
        abort();
    }

    /* Be sure the file was not tampered with while opening */

    /* fork process */

    pid = fork();
    if (pid == -1)
    {
        /* Handle error */
        (void)close(fd);
        abort();
    }
    else if (pid == 0)
    {  /* Close file descriptor in child process and repoen
        it in read only mode */

        (void)close(fd);
        fd = open(test_file, O_RDONLY);
        if (fd == -1)
        {
            /* Handle error */
            abort();
        }


        (void)read(fd, &c, 1);
        (void)close(fd);
    }
    else
    {  /* Parent acceses original file descriptor */
        (void)read(fd, &c, 1);
        (void)close(fd);
    }
}
```

Check Information

Group: Rule 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

POS38-C

Introduced in R2019a

CERT C: Rule POS39-C

Use the correct byte ordering when transferring data between systems

Description

Rule Definition

Use the correct byte ordering when transferring data between systems.

Polyspace Implementation

This checker checks for **Missing byte reordering when transferring data**.

Examples

Missing byte reordering when transferring data

Issue

Missing byte reordering when transferring data occurs when you do not use a byte ordering function:

- Before sending data to a network socket.
- After receiving data from a network socket.

Risk

Some system architectures implement little endian byte ordering (least significant byte first), and other systems implement big endian (most significant byte first). If the endianness of the sent data does not match the endianness of the receiving system, the value returned when reading the data is incorrect.

Fix

After receiving data from a socket, use a byte ordering function such as `ntohl()`. Before sending data to a socket, use a byte ordering function such as `htonl()` .

Example - Data Transferred Without Byte Reordering

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <byteswap.h>
#include <unistd.h>
#include <string.h>


unsigned int func(int sock, int server)
{
    unsigned int num;   /* assume int is 32-bits */
```

```
    if (server)
    {
        /* Server side */
        num = 0x17;
        /* Endianness of server host may not match endianness of network. */
        if (send(sock, (void *)&num, sizeof(num), 0) < (int)sizeof(num))
        {
            /* Handle error */
        }
        return 0;
    }
    else {
        /* Endianness of client host may not match endianness of network. */
        if (recv (sock, (void *)&num, sizeof(num), 0) < (int) sizeof(num))
        {
            /* Handle error */
        }

        /* Comparison may be inaccurate */
        if (num> 255)
        {
            return 255;
        }
        else
        {
            return num;
        }
    }
}
```

In this example, variable `num` is assigned hexadecimal value `0x17` and is sent over a network to the client from the server. If the server host is little endian and the network is big endian, `num` is transferred as `0x17000000`. The client then reads an incorrect value for `num` and compares it to a local numeric value.

Correction — Use Byte Ordering Function

Before sending `num` from the server host, use `htonl()` to convert from host to network byte ordering. Similarly, before reading `num` on the client host, use `ntohl()` to convert from network to host byte ordering.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <byteswap.h>
#include <unistd.h>
#include <string.h>

unsigned int func(int sock, int server)
{
    unsigned int num;   /* assume int is 32-bits */
    if (server)
    {
```

```
        /* Server side */
        num = 0x17;

        /* Convert to network byte order. */
        num = htonl(num);
        if (send(sock, (void *)&num, sizeof(num), 0) < (int)sizeof(num))
        {
            /* Handle error */
        }
        return 0;
    }
    else {
        if (recv (sock, (void *)&num, sizeof(num), 0) < (int) sizeof(num))
        {
            /* Handle error */
        }

        /* Convert to host byte order. */
        num = ntohl(num);
        if (num > 255)
        {
            return 255;
        }
        else
        {
            return num;
        }
    }
}
```

Check Information

Group: Rule 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

POS39-C

Introduced in R2019a

CERT C: Rule POS44-C

Do not use signals to terminate threads

Description

Rule Definition

Do not use signals to terminate threads.

Polyspace Implementation

This checker checks for **Use of signal to kill thread**.

Examples

Use of signal to kill thread

Issue

Use of signal to kill thread occurs when you use an uncaught signal to kill a thread. For instance, you use the POSIX function `pthread_kill` and send the signal `SIGTERM` to kill a thread.

Risk

Sending a signal kills the entire process instead of just the thread that you intend to kill.

For instance, the `pthread_kill` specifications state that if the disposition of a signal is to terminate, this action affects the entire process.

Fix

Use other mechanisms that are intended to kill specific threads.

For instance, use the POSIX function `pthread_cancel` to terminate a specific thread.

Example - Use of `pthread_kill` to Terminate Threads

```
#include <signal.h>
#include <pthread.h>

void* func(void *foo) {
  /* Execution of thread */
}

int main(void) {
  int result;
  pthread_t thread;

  if ((result = pthread_create(&thread, NULL, func, 0)) != 0) {
  }
  if ((result = pthread_kill(thread, SIGTERM)) != 0) {
  }

  /* This point is not reached because the process terminates in pthread_kill() */
```

```
  return 0;
}
```

In this example, the `pthread_kill` function sends the signal `SIGTERM` to kill a thread. The signal kills the entire process instead of the thread previously created with `pthread_create`.

Correction — Use `pthread_cancel` to Terminate Threads

One possible correction is to use the `pthread_cancel` function. The `pthread_cancel` terminates a thread specified by its first argument at a specific cancellation point or immediately, depending on the thread's cancellation type.

```
#include <signal.h>
#include <pthread.h>

void* func(void *foo) {
  /* Execution of thread */
}

int main(void) {
  int result;
  pthread_t thread;

  if ((result = pthread_create(&thread, NULL, func, 0)) != 0) {
    /* Handle Error */
  }
  if ((result = pthread_cancel(thread)) != 0) {
    /* Handle Error */
  }

  /* Continue executing */

  return 0;
}
```

See also:

- `pthread_cancel` for more information on cancellation types.
- Pthreads for functions that are allowed to be cancellation points.

Check Information

Group: Rule 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

POS44-C

Introduced in R2019a

CERT C: Rule POS47-C

Do not use threads that can be canceled asynchronously

Description

Rule Definition

Do not use threads that can be canceled asynchronously.

Polyspace Implementation

This checker checks for **Asynchronously cancellable thread**.

Examples

Asynchronously cancellable thread

Issue

This issue occurs when you use `pthread_setcanceltype` with argument `PTHREAD_CANCEL_ASYNCHRONOUS` to set the cancellability type of a calling thread to asynchronous (or immediate) . An asynchronously cancellable thread can be cancelled at any time, usually immediately upon receiving a cancellation request.

Risk

The calling thread might be cancelled in an unsafe state that could result in a resources leak, a deadlock, a data race, data corruption, or unpredictable behavior.

Fix

Remove the call to `pthread_setcanceltype` with argument `PTHREAD_CANCEL_ASYNCHRONOUS` to use the default cancellability type `PTHREAD_CANCEL_DEFERRED` instead. With the default cancellability type, the thread defers cancellation requests until it calls a function that is a cancellation point.

Example - Cancellability Type of Thread Set to Asynchronous

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static int fatal_error(void)
{
    exit(1);
}


volatile int a = 5;
volatile int b = 10;

pthread_mutex_t global_lock = PTHREAD_MUTEX_INITIALIZER;

void* swap_values_thread(void* dummy)
{
    int i;
    int c;
    int result;
    if ((result =
             pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &i)) != 0) {
```

```
        /* handle error */
        fatal_error();
    }
    while (1) {
        if ((result = pthread_mutex_lock(&global_lock)) != 0) {
            /* handle error */
            fatal_error();
        }
        c = b;
        b = a;
        a = c;
        if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
            /* handle error */
            fatal_error();
        }
    }
    return NULL;
}

int main(void)
{
    int result;
    pthread_t worker;

    if ((result = pthread_create(&worker, NULL, swap_values_thread, NULL)) != 0) {
        /* handle error */
        fatal_error();
    }

    /* Additional code */

    if ((result = pthread_cancel(worker)) != 0) {
        /* handle error */
        fatal_error();
    }


    if ((result = pthread_join(worker, 0)) != 0) {
        /* handle error */
        fatal_error();
    }

    if ((result = pthread_mutex_lock(&global_lock)) != 0) {
        /* handle error */
        fatal_error();
    }
    printf("a: %i | b: %i", a, b);
    if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
        /* handle error */
        fatal_error();
    }

    return 0;
}
```

In this example, the cancellability type of the `worker` thread is set to asynchronous. The mutex `global_lock` helps ensure that the `worker` and `main` threads do not access variables `a` and `b` at the same time. However, the `worker` thread might be cancelled while holding `global_lock`, and the `main` thread will never acquire `global_lock`, which results in a deadlock.

Correction — Use the Default Cancellability Type

One possible correction is to remove the call to `pthread_setcanceltype`. By default, the cancellability type of a new thread is set to `PTHREAD_CANCEL_DEFERRED`. The `worker` thread defers cancellation requests until it calls a function that is a cancellation point.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static int fatal_error(void)
{
    exit(1);
}


volatile int a = 5;
```

```
volatile int b = 10;

pthread_mutex_t global_lock = PTHREAD_MUTEX_INITIALIZER;

void* swap_values_thread(void* dummy)
{
    int i;
    int c;
    int result;
    while (1) {
        if ((result = pthread_mutex_lock(&global_lock)) != 0) {
            /* handle error */
            fatal_error();
        }
        c = b;
        b = a;
        a = c;
        if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
            /* handle error */
            fatal_error();
        }
    }
    return NULL;
}

int main(void)
{
    int result;
    pthread_t worker;

    if ((result = pthread_create(&worker, NULL, swap_values_thread, NULL)) != 0) {
        /* handle error */
        fatal_error();
    }

    /* Additional code */

    if ((result = pthread_cancel(worker)) != 0) {
        /* handle error */
        fatal_error();
    }


    if ((result = pthread_join(worker, 0)) != 0) {
        /* handle error */
        fatal_error();
    }

    if ((result = pthread_mutex_lock(&global_lock)) != 0) {
        /* handle error */
        fatal_error();
    }
    printf("a: %i | b: %i", a, b);
    if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
        /* handle error */
        fatal_error();
    }

    return 0;
}
```

Check Information

Group: Rule 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

POS47-C

Introduced in R2020a

CERT C: Rule POS48-C

Do not unlock or destroy another POSIX thread's mutex

Description

Rule Definition

Do not unlock or destroy another POSIX thread's mutex.

Polyspace Implementation

This checker checks for **Destruction of locked mutex**.

Examples

Destruction of locked mutex

Issue

Destruction of locked mutex occurs when a task destroys a mutex after it is locked (and before it is unlocked). The locking and destruction can happen in the same task or different tasks.

Risk

A mutex is locked to protect shared variables from concurrent access. If a mutex is destroyed in the locked state, the protection does not apply.

Fix

To fix this defect, destroy the mutex only after you unlock it. It is a good design practice to:

- Initialize a mutex *before* creating the threads where you use the mutex.
- Destroy a mutex *after* joining the threads that you created.

On the **Result Details** pane, you see two events, the locking and destruction of the mutex, and the tasks that initiated the events. To navigate to the corresponding line in your source code, click the event.

Example - Locking and Destruction in Different Tasks

```
#include <pthread.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;
pthread_mutex_t lock3;

void t0 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
  pthread_mutex_lock (&lock3);
  pthread_mutex_unlock (&lock2);
```

```
  pthread_mutex_unlock (&lock1);
  pthread_mutex_unlock (&lock3);
}

void t1 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
  pthread_mutex_destroy (&lock3);
  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}
```

In this example, after task `t0` locks the mutex `lock3`, task `t1` can destroy it. The destruction occurs if the following events happen in sequence:

1. `t0` acquires `lock3`.
2. `t0` releases `lock2`.
3. `t0` releases `lock1`.
4. `t1` acquires the lock `lock1` released by `t0`.
5. `t1` acquires the lock `lock2` released by `t0`.
6. `t1` destroys `lock3`.

For simplicity, this example uses a mix of automatic and manual concurrency detection. The tasks `t0` and `t1` are manually specified as entry points by using the option `Tasks (-entry-points)`. The critical sections are implemented through primitives `pthread_mutex_lock` and `pthread_mutex_unlock` that the software detects automatically. In practice, for entry point specification (thread creation), you will use primitives such as `pthread_create`. The next example shows how the defect can appear when you use `pthread_create`.

Correction — Place Lock-Unlock Pair Together in Same Critical Section as Destruction

The locking and destruction of `lock3` occurs inside the critical section imposed by `lock1` and `lock2`, but the unlocking occurs outside. One possible correction is to place the lock-unlock pair in the same critical section as the destruction of the mutex. Use one of these critical sections:

- Critical section imposed by `lock1` alone.
- Critical section imposed by `lock1` and `lock2`.

In this corrected code, the lock-unlock pair and the destruction is placed in the critical section imposed by `lock1` and `lock2`. When `t0` acquires `lock1` and `lock2`, `t1` has to wait for their release before it executes the instruction `pthread_mutex_destroy (&lock3);`. Therefore, `t1` cannot destroy mutex `lock3` in the locked state.

```
#include <pthread.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;
pthread_mutex_t lock3;

void t0 (void) {
  pthread_mutex_lock (&lock1);
```

```
  pthread_mutex_lock (&lock2);

  pthread_mutex_lock (&lock3);
  pthread_mutex_unlock (&lock3);

  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}

void t1 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);

  pthread_mutex_destroy (&lock3);

  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}
```

Example - Locking and Destruction in Start Routine of Thread

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 4
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
void atomic_operation(void);

void *do_create(void *arg) {
    /* Creation thread */
    pthread_mutex_init(&lock, NULL);
    pthread_exit((void*) 0);
}

void *do_work(void *arg) {
    /* Worker thread */
    pthread_mutex_lock (&lock);
    atomic_operation();
    pthread_mutex_unlock (&lock);
    pthread_exit((void*) 0);
}

void *do_destroy(void *arg) {
    /* Destruction thread */
    pthread_mutex_destroy(&lock);
    pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
```

```
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Thread that initializes mutex */
   pthread_create(&callThd[0], &attr, do_create, NULL);

   /* Threads that use mutex for atomic operation*/
   for(i=0; i<NUMTHREADS-1; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   /* Thread that destroys mutex */
   pthread_create(&callThd[NUMTHREADS -1], &attr, do_destroy, NULL);

   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   pthread_exit(NULL);
}
```

In this example, four threads are created. The threads are assigned different actions.

- The first thread `callThd[0]` initializes the mutex `lock`.
- The second and third threads, `callThd[1]` and `callThd[2]`, perform an atomic operation protected by the mutex `lock`.
- The fourth thread `callThd[3]` destroys the mutex `lock`.

The threads can interrupt each other. Therefore, immediately after the second or third thread locks the mutex, the fourth thread can destroy it.

Correction — Initialize and Destroy Mutex Outside Start Routine

One possible correction is to initialize and destroy the mutex in the `main` function outside the start routine of the threads. The threads perform only the atomic operation. You need two fewer threads because the mutex initialization and destruction threads are not required.

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 2
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
void atomic_operation(void);

void *do_work(void *arg) {
   pthread_mutex_lock (&lock);
   atomic_operation();
   pthread_mutex_unlock (&lock);
   pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   int i;
   void *status;
```

```
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Initialize mutex */
   pthread_mutex_init(&lock, NULL);

   for(i=0; i<NUMTHREADS; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* Destroy mutex */
   pthread_mutex_destroy(&lock);

   pthread_exit(NULL);
}
```

Correction — Use A Second Mutex To Protect Lock-Unlock Pair and Destruction

Another possible correction is to use a second mutex and protect the lock-unlock pair from the destruction. This corrected code uses the mutex `lock2` to achieve this protection. The second mutex is initialized in the `main` function outside the start routine of the threads.

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 4
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
pthread_mutex_t lock2;
void atomic_operation(void);

void *do_create(void *arg) {
    /* Creation thread */
    pthread_mutex_init(&lock, NULL);
    pthread_exit((void*) 0);
}

void *do_work(void *arg) {
    /* Worker thread */
    pthread_mutex_lock (&lock2);
    pthread_mutex_lock (&lock);
    atomic_operation();
    pthread_mutex_unlock (&lock);
    pthread_mutex_unlock (&lock2);
    pthread_exit((void*) 0);
}
```

```
void *do_destroy(void *arg) {
    /* Destruction thread */
    pthread_mutex_lock (&lock2);
    pthread_mutex_destroy(&lock);
    pthread_mutex_unlock (&lock2);
    pthread_exit((void*) 0);
}


int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Initialize second mutex */
   pthread_mutex_init(&lock2, NULL);

   /* Thread that initializes first mutex */
   pthread_create(&callThd[0], &attr, do_create, NULL);

   /* Threads that use first mutex for atomic operation */
   /* The threads use second mutex to protect first from destruction in locked state*/
   for(i=0; i<NUMTHREADS-1; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   /* Thread that destroys first mutex */
   /* The thread uses the second mutex to prevent destruction of locked mutex */
   pthread_create(&callThd[NUMTHREADS -1], &attr, do_destroy, NULL);


   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* Destroy second mutex */
   pthread_mutex_destroy(&lock2);

   pthread_exit(NULL);
}
```

Check Information

Group: Rule 50. POSIX (POS)

See Also

Check SEI CERT-C (-cert-c)

Topics
“Check for Coding Standard Violations”

External Websites
POS48-C

Introduced in R2019a

CERT C: Rule POS49-C

When data must be accessed by multiple threads, provide a mutex and guarantee no adjacent data is also accessed

Description

Rule Definition

When data must be accessed by multiple threads, provide a mutex and guarantee no adjacent data is also accessed.

Polyspace Implementation

This checker checks for **Data race on adjacent bit fields**.

Examples

Data race on adjacent bit fields

Issue

Data race on adjacent bit fields occurs when:

1. Multiple tasks perform unprotected operations on bit fields that are part of the same structure.

   For instance, a task operates on field `errorFlag1` and another task on field `errorFlag2` in a variable of this type:

   ```
   struct errorFlags {
      unsigned int errorFlag1 : 1;
      unsigned int errorFlag2 : 1;
      ...
   }
   ```

   Suppose that the operations are not atomic with respect to each other. In other words, you have not implemented protection mechanisms to ensure that one operation completes before another begins.
2. At least one of the unprotected operations is a write operation.

Risk

Adjacent bit fields that are part of the same structure might be stored in one byte in the same memory location. Read or write operations on all variables including bit fields happen one byte or word at a time. To modify only specific bits in a byte, steps similar to this happen in sequence:

1. The byte is loaded into RAM.
2. A mask is created so that only specific bits would be modified to the intended value and the remaining bits remain unchanged.
3. A bitwise OR operation is performed between the copy of the byte in RAM and the mask.
4. The byte with specific bits modified is copied back from RAM.

If two different bit fields are accessed, these four steps have to be performed for each bit field. If the accesses are not protected, all four steps for one bit field might not complete before the four steps for the other begin. As a result, the modification of one bit field might undo the modification of an adjacent bit field. For instance, the modification of `errorFlag1` and `errorFlag2` can happen in the following sequence. Steps marked 1 relate to modification of `errorFlag1` and steps marked 2 relate to that of `errorFlag2`.

1a. The byte with both `errorFlag1` and `errorFlag2` unmodified is copied into RAM, for purposes of modifying `errorFlag1`.

1b. A mask that modifies only `errorFlag1` is bitwise OR-ed with this copy.

2a. The byte containing both `errorFlag1` and `errorFlag2` unmodified is copied into RAM a second time, for purposes of modifying `errorFlag2`.

2b. A mask that modifies only `errorFlag2` is bitwise OR-ed with this second copy.

1c. The version with `errorFlag1` modified is copied back. This version has `errorFlag2` unmodified.

2c The version with `errorFlag2` modified is copied back. This version has `errorFlag1` unmodified and overwrites the previous modification.

Fix

To fix this defect, protect the operations on bit fields that are part of the same structure using critical sections, temporal exclusion or another means. See “Protections for Shared Variables in Multitasking Code”.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the  icon. For an example, see below.

Example - Unprotected Operation on Global Variable from Multiple POSIX Threads

```
#include <stdlib.h>
#include <pthread.h>
#define thread_success 0

typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void* task1 (void* arg) {
    InterruptConfigbitsProc12.IOFlag = 0;
    //Additional code
```

```
}

void* task2 (void* arg) {
    InterruptConfigbitsProc12.SetupFlag = 0;
    //Additional code
}

void main() {
    pthread_t thread1, thread2;
    if(thread_success != pthread_create(&thread1, NULL, task1, NULL)){
        //Handle error
    }
    if(thread_success != pthread_create(&thread2, NULL, task2, NULL)){
        //Handle error
    }
}
```

In this example, the threads with id `thread1` and `thread2` access different bit fields `IOFlag` and `SetupFlag`, which belong to the same structured variable `InterruptConfigbitsProc12`.

Correction - Use Critical Sections

One possible correction is to wrap the bit field accesses in a critical section. A critical section lies between a call to a lock function and an unlock function. In this correction, the critical section lies between the calls to functions `pthread_mutex_lock` and `pthread_mutex_unlock`.

```
#include <stdlib.h>
#include <pthread.h>
#define thread_success 0
#define lock_success 0

pthread_mutex_t lock;

typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void* task1 (void* arg) {
    if( lock_success != pthread_mutex_lock(&lock)) {
        //Handle error
    }
    InterruptConfigbitsProc12.IOFlag = 0;
    if( lock_success != pthread_mutex_unlock(&lock)) {
        //Handle error
    }
    //Additional code
}
```

```
void* task2 (void* arg) {
    if( lock_success != pthread_mutex_lock(&lock)) {
        //Handle error
    }
    InterruptConfigbitsProc12.SetupFlag = 0;
    if( lock_success != pthread_mutex_unlock(&lock)) {
        //Handle error
    }
    //Additional code
}

void main() {
    pthread_t thread1, thread2;
    if(thread_success != pthread_create(&thread1, NULL, task1, NULL)){
        //Handle error
    }
    if(thread_success != pthread_create(&thread2, NULL, task2, NULL)){
        //Handle error
    }
}
```

Correction – Insert Bit Field of Size 0

You can enter a non bit field member or an unnamed bit field member of size 0 in between two adjacent bit fields that might be accessed concurrently. A non bit field member or size 0 bit field member ensures that the subsequent bit field starts from a new memory location. In this corrected example, the size 0 bit field member ensures that `IOFlag` and `SetupFlag` are stored in distinct memory locations.

```
#include <stdlib.h>
#include <pthread.h>
#define thread_success 0

typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int : 0;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void* task1 (void* arg) {
    InterruptConfigbitsProc12.IOFlag = 0;
    //Additional code
}

void* task2 (void* arg) {
    InterruptConfigbitsProc12.SetupFlag = 0;
    //Additional code
}
```

```
void main() {
    pthread_t thread1, thread2;
    if(thread_success != pthread_create(&thread1, NULL, task1, NULL)){
        //Handle error
    }
    if(thread_success != pthread_create(&thread2, NULL, task2, NULL)){
        //Handle error
    }
}
```

Check Information

Group: Rule 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

POS49-C

Introduced in R2019a

CERT C: Rule POS50-C

Declare objects shared between POSIX threads with appropriate storage durations

Description

Rule Definition

Declare objects shared between POSIX threads with appropriate storage durations.

Examples

Automatic or thread local variable escaping from a POSIX thread

Issue

Automatic or thread local variable escaping from a POSIX thread occurs when an automatic or thread local variable is passed by address from one POSIX thread to another without ensuring that the variable stays alive through the duration of the latter thread.

Risk

An automatic or thread local variable is allocated on the stack at the beginning of a thread and its lifetime extends till the end of the thread. The variable is not guaranteed to be alive when a different thread accesses it.

For instance, consider the start function of a POSIX thread with these lines:

```
int start_thread(pthread_t *tid) {
   int aVar = 0;
   if(thrd_success != pthread_create(tid, NULL, start_thread_child, &aVar) {
     ...
   }
}
```

The `pthread_create` function creates a child thread with start function `start_thread_child` and passes the address of the automatic variable `aVar`to this function. When this child thread accesses `aVar`, the parent thread might have completed execution and `aVar` is no longer on the stack. The access might result in reading unpredictable values.

Fix

When you pass a variable from one thread to another, make sure that the variable lifetime matches or exceeds the lifetime of both threads. You can achieve this synchronization in one of these ways:

- Declare the variable `static` so that it does not go out of stack when the current thread completes execution.
- Dynamically allocate the storage for the variable so that it is allocated on the heap instead of the stack and must be explicitly deallocated. Make sure that the deallocation happens after both threads complete execution.

These solutions require you to create a variable in nonlocal memory. Instead, you can use other solutions such as the `shared` keyword with OpenMP's threading interface that allows you to safely share local variables across threads.

Example – Local Variable Escaping Thread

```
#include <pthread.h>
#include <stdio.h>

void* create_child_thread(void *childVal) {
  int *res = (int *)childVal;
  printf("Result: %d\n", *res);
  return NULL;
}

void create_parent_thread(pthread_t *tid) {
  int parentVal = 1;
  int thrd_success;
  if ((thrd_success = pthread_create(tid, NULL, create_child_thread, &parentVal)) != 0) {
    /* Handle error */
  }
}

int main(void) {
  pthread_t tid;
  int thrd_success;
  create_parent_thread(&tid);

  if ((thrd_success = thrd_join(tid, NULL)) != 0) {
    /* Handle error */
  }
  return 0;
}
```

In this example, the value `parentVal` is local to the parent thread that starts in `main` and continues into the function `create_parent_thread`. However, in the body of `create_parent_thread`, the address of this local variable is passed to a child thread (the thread with start routine `create_child_thread`). The parent thread might have completed execution and the variable `parentVal` might have gone out of scope when the child thread accesses this variable.

Correction – Use Static Variables

One possible correction is to declare the variable `parentVal` as `static` so that the variable is on the stack for the entire duration of the program.

```
#include <pthread.h>
#include <stdio.h>

void* create_child_thread(void *childVal) {
  int *res = (int *)childVal;
  printf("Result: %d\n", *res);
  return NULL;
}

void create_parent_thread(pthread_t *tid) {
  static int parentVal = 1;
  int thrd_success;
  if ((thrd_success = pthread_create(tid, NULL, create_child_thread, &parentVal)) != 0) {
    /* Handle error */
  }
}
```

```
int main(void) {
  pthread_t tid;
  int thrd_success;
  create_parent_thread(&tid);

  if ((thrd_success = thrd_join(tid, NULL)) != 0) {
    /* Handle error */
  }
  return 0;
}
```

Correction – Use Dynamic Memory Allocation

One possible correction is to dynamically allocate storage for variables to be shared across threads and explicitly free the storage after the variable is no longer required.

```
#include <pthread.h>
#include <stlib.h>

void* create_child_thread(void *val) {
  int *res = (int *)val;
  printf("Result: %d\n", *res);
  free(res);
  return NULL;
}

void create_parent_thread(pthread_t *tid) {
  int *val;
  int thrd_success;

  val = malloc(sizeof(int));

  if(!val) {
      *val = 1;
      if ((thrd_success = pthread_create(tid, NULL, create_child_thread, val)) != 0) {
        /* Handle error */
      }
  }
}

int main(void) {
  pthread_t tid;
  int thrd_success;
  create_parent_thread(&tid);

  if ((thrd_success = thrd_join(tid, NULL)) != 0) {
    /* Handle error */
  }
  return 0;
}
```

Check Information

Group: Rule 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics
“Check for Coding Standard Violations”

External Websites
POS50-C

Introduced in R2020a

CERT C: Rule POS51-C

Avoid deadlock with POSIX threads by locking in predefined order

Description

Rule Definition

Avoid deadlock with POSIX threads by locking in predefined order.

Polyspace Implementation

This checker checks for **Deadlock**.

Examples

Deadlock

Issue

Deadlock occurs when multiple tasks are stuck in their critical sections (CS) because:

- Each CS waits for another CS to end.
- The critical sections (CS) form a closed cycle. For example:
  - CS #1 waits for CS #2 to end, and CS #2 waits for CS #1 to end.
  - CS #1 waits for CS #2 to end, CS #2 waits for CS #3 to end and CS #3 waits for CS #1 to end.

Polyspace expects critical sections of code to follow a specific format. A critical section lies between a call to a lock function and a call to an unlock function. When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait until `my_task` calls the corresponding unlock function. Both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Risk

Each task waits for a critical section in another task to end and is unable to proceed. The program can freeze indefinitely.

Fix

The fix depends on the root cause of the defect. You can try to break the cyclic order between the tasks in one of these ways:

- Write down all critical sections involved in the deadlock in a certain sequence. Whenever you call the lock functions of the critical sections within a task, respect the order in that sequence. See an example below.
- If one of the critical sections involved in a deadlock occurs in an interrupt, try to disable all interrupts during critical sections in all tasks. See `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Reviewing this defect is an opportunity to check if all operations in your critical section are really meant to be executed as an atomic block. It is a good practice to keep critical sections at a bare minimum.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Deadlock with Two Tasks

```
void task1(void);
void task2(void);

int var;
void perform_task_cycle(void) {
 var++;
}

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_2();
    begin_critical_section_1();
    perform_task_cycle();
    end_critical_section_1();
    end_critical_section_2();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Specification
Configure multitasking manually	☑

Entry points	`task1`
	
	`task2`

Option	Specification	
Critical section details	**Starting routine**	**Ending routine**
	`begin_critical_section_1`	`end_critical_section_1`
	`begin_critical_section_2`	`end_critical_section_2`

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. `task1` calls `begin_critical_section_1`.
2. `task2` calls `begin_critical_section_2`.
3. `task1` reaches the instruction `begin_critical_section_2();`. Since `task2` has already called `begin_critical_section_2`, `task1` waits for `task2` to call `end_critical_section_2`.
4. `task2` reaches the instruction `begin_critical_section_1();`. Since `task1` has already called `begin_critical_section_1`, `task2` waits for `task1` to call `end_critical_section_1`.

Correction-Follow Same Locking Sequence in Both Tasks

One possible correction is to follow the same sequence of calls to lock and unlock functions in both `task1` and `task2`.

```
void task1(void);
void task2(void);
void perform_task_cycle(void);

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}
```

Example - Deadlock with More Than Two Tasks

```
int var;
void performTaskCycle() {
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);


void unlock1(void);
void unlock2(void);
void unlock3(void);

void task1() {
 while(1) {
    lock1();
    lock2();
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock3();
    lock1();
    performTaskCycle();
    unlock1();
    unlock3();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Specification
Configure multitasking manually	☑

Option	Specification	

Entry points	`task1`
	`task2`
	`task3`	

Critical section details	**Starting routine**	**Ending routine**
	`lock1`	`unlock1`
	`lock2`	`unlock2`
	`lock3`	`unlock3`

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. `task1` calls `lock1`.
2. `task2` calls `lock2`.
3. `task3` calls `lock3`.
4. `task1` reaches the instruction `lock2();`. Since `task2` has already called `lock2`, `task1` waits for call to `unlock2`.
5. `task2` reaches the instruction `lock3();`. Since `task3` has already called `lock3`, `task2` waits for call to `unlock3`.
6. `task3` reaches the instruction `lock1();`. Since `task1` has already called `lock1`, `task3` waits for call to `unlock1`.

Correction — Break Cyclic Order

To break the cyclic order between critical sections, note every lock function in your code in a certain sequence, for example:

1. `lock1`
2. `lock2`
3. `lock3`

If you use more than one lock function in a task, use them in the order in which they appear in the sequence. For example, you can use `lock1` followed by `lock2` but not `lock2` followed by `lock1`.

```
int var;
void performTaskCycle() {
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);

void unlock1(void);
void unlock2(void);
void unlock3(void);
```

```
void task1() {
 while(1) {
    lock1();
    lock2();
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock1();
    lock3();
    performTaskCycle();
    unlock3();
    unlock1();
 }
}
```

Check Information

Group: Rule 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

POS51-C

Introduced in R2019a

CERT C: Rule POS52-C

Do not perform operations that can block while holding a POSIX lock

Description

Rule Definition

Do not perform operations that can block while holding a POSIX lock.

Polyspace Implementation

This checker checks for **Blocking operation while holding lock**.

Examples

Blocking operation while holding lock

Issue

Blocking operation while holding lock occurs when a task (thread) performs a potentially lengthy operation while holding a lock.

The checker considers calls to these functions as potentially lengthy:

- Functions that access a network such as `recv`
- System call functions such as `fork`, `pipe` and `system`
- Functions for I/O operations such as `getchar` and `scanf`
- File handling functions such as `fopen`, `remove` and `lstat`
- Directory manipulation functions such as `mkdir` and `rmdir`

The checker automatically detects certain primitives that hold and release a lock, for instance, `pthread_mutex_lock` and `pthread_mutex_unlock`. For the full list of primitives that are automatically detected, see “Auto-Detection of Thread Creation and Critical Section in Polyspace”.

Risk

If a thread performs a lengthy operation when holding a lock, other threads that use the lock have to wait for the lock to be available. As a result, system performance can slow down or deadlocks can occur.

Fix

Perform the blocking operation before holding the lock or after releasing the lock.

Some functions detected by this checker can be called in a way that does not make them potentially lengthy. For instance, the function `recv` can be called with the parameter `O_NONBLOCK` which causes the call to fail if no message is available. When called with this parameter, `recv` does not wait for a message to become available.

Example - Network I/O Operations with `recv` While Holding Lock

```
#include <pthread.h>
#include <sys/socket.h>
```

```
pthread_mutexattr_t attr;
pthread_mutex_t mutex;

void thread_foo(void *ptr) {
  unsigned int num;
  int result;
  int sock;

  /* sock is a connected TCP socket */

  if ((result = pthread_mutex_lock(&mutex)) != 0) {
    /* Handle Error */
  }

  if ((result = recv(sock, (void *)&num, sizeof(unsigned int), 0)) < 0) {
    /* Handle Error */
  }

  /* ... */

  if ((result = pthread_mutex_unlock(&mutex)) != 0) {
    /* Handle Error */
  }
}

int main() {
  pthread_t thread;
  int result;

  if ((result = pthread_mutexattr_settype(
      &attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_init(&mutex, &attr)) != 0) {
    /* Handle Error */
  }

  if (pthread_create(&thread, NULL,(void*(*)(void*))& thread_foo, NULL) != 0) {
    /* Handle Error */
  }

  /* ... */

  pthread_join(thread, NULL);

  if ((result = pthread_mutex_destroy(&mutex)) != 0) {
    /* Handle Error */
  }

  return 0;
}
```

In this example, in each thread created with `pthread_create`, the function `thread_foo` performs a network I/O operation with `recv` after acquiring a lock with `pthread_mutex_lock`. Other threads

using the same lock variable `mutex` have to wait for the operation to complete and the lock to become available.

Correction — Perform Blocking Operation Before Acquiring Lock

One possible correction is to call `recv` before acquiring the lock.

```
#include <pthread.h>
#include <sys/socket.h>

pthread_mutexattr_t attr;
pthread_mutex_t mutex;

void thread_foo(void *ptr) {
  unsigned int num;
  int result;
  int sock;

  /* sock is a connected TCP socket */
  if ((result = recv(sock, (void *)&num, sizeof(unsigned int), 0)) < 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_lock(&mutex)) != 0) {
    /* Handle Error */
  }

    /* ... */

  if ((result = pthread_mutex_unlock(&mutex)) != 0) {
    /* Handle Error */
  }
}

int main() {
  pthread_t thread;
  int result;

  if ((result = pthread_mutexattr_settype(
      &attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_init(&mutex, &attr)) != 0) {
    /* Handle Error */
  }

  if (pthread_create(&thread, NULL,(void*(*)(void*))& thread_foo, NULL) != 0) {
    /* Handle Error */
  }

  /* ... */

  pthread_join(thread, NULL);

  if ((result = pthread_mutex_destroy(&mutex)) != 0) {
    /* Handle Error */
  }
```

```
  return 0;
}
```

Check Information

Group: Rule 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

POS52-C

Introduced in R2019a

CERT C: Rec. POS53-C

Do not use more than one mutex for concurrent waiting operations on a condition variable

Description

Rule Definition

Do not use more than one mutex for concurrent waiting operations on a condition variable.

Polyspace Implementation

This checker checks for **Multiple mutexes used with same conditional variable**.

Examples

Multiple mutexes used with same conditional variable

Issue

This issue occurs when multiple threads use more than one mutex to concurrently wait on the same condition variable. A thread waits on a condition variable by calling the functions `pthread_cond_timedwait` or `pthread_cond_wait`. These functions take a condition variable and a locked mutex as arguments, and the condition variable is bound to that mutex when the thread waits on the condition variable.

The checkers flags the use of `pthread_cond_timedwait` or `pthread_cond_wait` in one of the threads. See the **Event** column in the **Results Details** pane to view the threads waiting on the same condition variable and using a different mutex.

Risk

When a thread waits on a condition variable using a mutex, the condition variable is bound to that mutex. Any other thread using a different mutex to wait on the same condition variable is undefined behavior according to the POSIX standard.

Fix

Use the same mutex argument for `pthread_cond_timedwait` or `pthread_cond_wait` when threads are concurrently waiting on the same condition variable, or use separate condition variables for each mutex.

Example - Concurrent Waiting on Condition Variable with Multiple Mutexes

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define Thrd_return_t                   void *
#define __USE_XOPEN2K8



#define COUNT_LIMIT 5

static void fatal_error(void)
{
    exit(1);
}
```

```
pthread_mutex_t mutex1;
pthread_mutex_t mutex2;
pthread_mutex_t mutex3;
pthread_cond_t cv;

int count1 = 0, count2 = 0, count3 = 0;
#define DELAY 8

Thrd_return_t waiter1(void* arg)
{
    int ret;
    while (count1 < COUNT_LIMIT) {
        if ((ret = pthread_mutex_lock(&mutex1)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                  pthread_cond_wait(&cv, &mutex1)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count1 = %d\n", ++count1);
        if ((ret = pthread_mutex_unlock(&mutex1)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t waiter2(void* arg)
{
    int ret;
    while (count2 < COUNT_LIMIT) {
        if ((ret = pthread_mutex_lock(&mutex2)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                  pthread_cond_wait(&cv, &mutex2)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count2 = %d\n", ++count2);
        if ((ret = pthread_mutex_unlock(&mutex2)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t signaler(void* arg)
{
    int ret;
    while ((count1 < COUNT_LIMIT) || (count2 < COUNT_LIMIT)) {
        sleep(1);
        printf("signaling\n");
        if ((ret = pthread_cond_broadcast(&cv)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t waiter3(void* arg)
{
    int ret;
    while (count3 % COUNT_LIMIT != 0) {
        if ((ret = pthread_mutex_lock(&mutex3)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                  pthread_cond_wait(&cv, &mutex3)) != 0) {
            /* Handle error */
```

```
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count3 = %d\n", ++count3);
        if ((ret = pthread_mutex_unlock(&mutex3)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

int main(void)
{
    int ret;
    pthread_t thread1, thread2, thread3;

    pthread_mutexattr_t attr;

    if ((ret = pthread_mutexattr_init(&attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
        /* Handle error */
        fatal_error();
    }

    if ((ret = pthread_mutex_init(&mutex1, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutex_init(&mutex2, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutex_init(&mutex3, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_cond_init(&cv, NULL)) != 0) {
        /* handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread1, NULL, &waiter1, NULL))) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread2, NULL, &waiter2, NULL))) {
        /* handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread3, NULL, &signaler, NULL))) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread1, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread2, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread3, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }

    while (1) { ; }

    return 0;
}
```

In this example, a different mutex is used to protect each `count` variable. Since all three `waiter` functions wait on the same condition variable `cv` with different mutexes, the call to `pthread_cond_wait` will succeed for one of the threads and the call will be undefined for the other two.

The checker raises a defect for function `waiter3` even though the function is not invoked directly or indirectly by a thread, entry-point, or interrupt. The analysis considers function `waiter3` called by the main program through its function address or an unidentified thread whose creation is the missing source code.

Correction — Use the Same Mutex for All Threads Waiting on Same Condition Variable

One possible correction is to pass the same mutex argument to all the call to `pthread_cond_wait` that are used to wait on the same condition variable.

```
 #include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define Thrd_return_t                   void *
#define __USE_XOPEN2K8



#define COUNT_LIMIT 5

static void fatal_error(void)
{
    exit(1);
}


pthread_mutex_t mutex;

pthread_cond_t cv;

int count1 = 0, count2 = 0, count3 = 0;
#define DELAY 8

Thrd_return_t waiter1(void* arg)
{
    int ret;
    while (count1 < COUNT_LIMIT) {
        if ((ret = pthread_mutex_lock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                  pthread_cond_wait(&cv, &mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count1 = %d\n", ++count1);
        if ((ret = pthread_mutex_unlock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t waiter2(void* arg)
{
    int ret;
    while (count2 < COUNT_LIMIT) {
        if ((ret = pthread_mutex_lock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                  pthread_cond_wait(&cv, &mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count2 = %d\n", ++count2);
        if ((ret = pthread_mutex_unlock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
```

```
}

Thrd_return_t signaler(void* arg)
{
    int ret;
    while ((count1 < COUNT_LIMIT) || (count2 < COUNT_LIMIT)) {
        sleep(1);
        printf("signaling\n");
        if ((ret = pthread_cond_broadcast(&cv)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t waiter3(void* arg)
{
    int ret;
    while (count3 % COUNT_LIMIT != 0) {
        if ((ret = pthread_mutex_lock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                pthread_cond_wait(&cv, &mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count3 = %d\n", ++count3);
        if ((ret = pthread_mutex_unlock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}
/*
void user_task(void)
{
    (void)waiter3(NULL);
} */

int main(void)
{
    int ret;
    pthread_t thread1, thread2, thread3;

    pthread_mutexattr_t attr;

    if ((ret = pthread_mutexattr_init(&attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
        /* Handle error */
        fatal_error();
    }

    if ((ret = pthread_mutex_init(&mutex, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutex_init(&mutex, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutex_init(&mutex, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_cond_init(&cv, NULL)) != 0) {
        /* handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread1, NULL, &waiter1, NULL))) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread2, NULL, &waiter2, NULL))) {
```

```
        /* handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread3, NULL, &signaler, NULL))) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread1, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread2, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread3, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }

    while (1) { ; }

    return 0;
}
```

Check Information

Group: Rule 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

POS53-C

Introduced in R2020a

CERT C: Rule POS54-C

Detect and handle POSIX library errors

Description

Rule Definition

Detect and handle POSIX library errors.

Polyspace Implementation

This checker checks for **Returned value of a sensitive function not checked**.

Examples

Returned value of a sensitive function not checked

Issue

Returned value of a sensitive function not checked occurs when you call sensitive standard functions, but you:

- Ignore the return value.
- Use an output or a return value without testing the validity of the return value.

For this defect, two type of functions are considered: ***sensitive*** and ***critical sensitive***.

A ***sensitive*** function is a standard function that can encounter:

- Exhausted system resources (for example, when allocating resources)
- Changed privileges or permissions
- Tainted sources when reading, writing, or converting data from external sources
- Unsupported features despite an existing API

A ***critical sensitive*** function is a sensitive function that performs one of these critical or vulnerable tasks:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

Risk

If you do not check the return value of functions that perform sensitive or critical sensitive tasks, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

Fix

Before continuing with the program, test the return value of *critical sensitive* functions.

For *sensitive functions*, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

Example - Sensitive Function Return Ignored

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    pthread_attr_init(&attr);
}
```

This example shows a call to the sensitive function `pthread_attr_init`. The return value of `pthread_attr_init` is ignored, causing a defect.

Correction — Cast Function to `(void)`

One possible correction is to cast the function to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of the sensitive function.

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    (void)pthread_attr_init(&attr);
}
```

Correction — Test Return Value

One possible correction is to test the return value of `pthread_attr_init` to check for errors.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

void initialize() {
    pthread_attr_t attr;
    int result;

    result = pthread_attr_init(&attr);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

Example - Critical Function Return Ignored

```
#include <pthread.h>
extern void *start_routine(void *);

void returnnotchecked() {
```

```
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0));
    pthread_join(thread_id,  &res);
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), Polyspace does not ignore this *Return value of a sensitive function not checked* defect. The other critical function, `pthread_join`, returns value that is ignored implicitly. `pthread_join` uses the return value of `pthread_create`, which was not checked.

Correction — Test the Return Value of Critical Functions

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

Check Information

Group: Rule 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites
POS54-C

Introduced in R2019a

CERT C: Rule WIN30-C

Properly pair allocation and deallocation functions

Description

Rule Definition

Properly pair allocation and deallocation functions.

Polyspace Implementation

This checker checks for **Mismatched alloc/dealloc functions on Windows**.

Examples

Mismatched alloc/dealloc functions on Windows

Issue

Mismatched alloc/dealloc functions on Windows occurs when you use a Windows deallocation function that is not properly paired to its corresponding allocation function.

Risk

Deallocating memory with a function that does not match the allocation function can cause memory corruption or undefined behavior. If you are using an older version of Windows, the improper function can also cause compatibility issues with newer versions.

Fix

Properly pair your allocation and deallocation functions according to the functions listed in this table.

Allocation Function	Deallocation Function
`malloc()`	`free()`
`realloc()`	`free()`
`calloc()`	`free()`
`_aligned_malloc()`	`_aligned_free()`
`_aligned_offset_malloc()`	`_aligned_free()`
`_aligned_realloc()`	`_aligned_free()`
`_aligned_offset_realloc()`	`_aligned_free()`
`_aligned_recalloc()`	`_aligned_free()`
`_aligned_offset_recalloc()`	`_aligned_free()`
`_malloca()`	`_freea()`
`LocalAlloc()`	`LocalFree()`
`LocalReAlloc()`	`LocalFree()`
`GlobalAlloc()`	`GlobalFree()`

Allocation Function	Deallocation Function
GlobalReAlloc()	GlobalFree()
VirtualAlloc()	VirtualFree()
VirtualAllocEx()	VirtualFreeEx()
VirtualAllocExNuma()	VirtualFreeEx()
HeapAlloc()	HeapFree()
HeapReAlloc()	HeapFree()

Example - Memory Deallocated with Incorrect Function

```
#ifdef _WIN32_
#include <windows.h>
#else
#define _WIN32_
typedef void *HANDLE;
typedef HANDLE HGLOBAL;
typedef HANDLE HLOCAL;
typedef unsigned int UINT;
extern HLOCAL LocalAlloc(UINT uFlags, UINT uBytes);
extern HLOCAL LocalFree(HLOCAL hMem);
extern HGLOBAL GlobalFree(HGLOBAL hMem);
#endif

#define SIZE9 9


void func(void)
{
    /* Memory allocation */
    HLOCAL p = LocalAlloc(0x0000, SIZE9);

    if (p) {
        /* Memory deallocation. */
        GlobalFree(p);

    }
}
```

In this example, memory is allocated with `LocallAlloc()`. The program then erroneously uses `GlobalFree()` to deallocate the memory.

Correction — Properly Pair Windows Allocation and Deallocation Functions

When you allocate memory with `LocalAllocate()`, use `LocalFree()` to deallocate the memory.

```
#ifdef _WIN32_
#include <windows.h>
#else
#define _WIN32_
typedef void *HANDLE;
typedef HANDLE HGLOBAL;
typedef HANDLE HLOCAL;
typedef unsigned int UINT;
```

```
extern HLOCAL LocalAlloc(UINT uFlags, UINT uBytes);
extern HLOCAL LocalFree(HLOCAL hMem);
extern HGLOBAL GlobalFree(HGLOBAL hMem);
#endif

#define SIZE9 9
void func(void)
{
    /* Memory allocation */
    HLOCAL p = LocalAlloc(0x0000, SIZE9);
    if (p) {
        /* Memory deallocation. */
        LocalFree(p);
    }
}
```

Check Information

Group: Rule 51. Microsoft Windows (WIN)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

WIN30-C

Introduced in R2019a

CERT C: Rec. PRE00-C

Prefer inline or static functions to function-like macros

Description

Rule Definition

Prefer inline or static functions to function-like macros.

Polyspace Implementation

This checker checks for **Use of function-like macro instead of function**.

Examples

Use of function-like macro instead of function

Issue

The issue occurs when you use a function-like macro instead of a function when the two are interchangeable.

Polyspace considers all function-like macro definitions.

Risk

In most circumstances, use functions instead of macros. Functions perform argument type-checking and evaluate their arguments once, avoiding problems with potential multiple side effects.

Check Information

Group: Rec. 01. Preprocessor (PRE)

See Also

`Check SEI CERT-C (-cert-c)`

Topics
“Check for Coding Standard Violations”

External Websites
PRE00-C

Introduced in R2019a

CERT C: Rec. PRE01-C

Use parentheses within macros around parameter names

Description

Rule Definition

Use parentheses within macros around parameter names.

Polyspace Implementation

This checker checks for **Expanded macro parameters not enclosed in parentheses**.

Examples

Expanded macro parameters not enclosed in parentheses

Issue

The issue occurs when expressions resulting from the expansion of macro parameters are not enclosed in parentheses.

Risk

If you do not use parentheses, then it is possible that operator precedence does not give the results that you want when macro substitution occurs.

If you are not using a macro parameter as an expression, then the parentheses are not necessary because no operators are involved in the macro.

Example - Macro Expressions

```
#define mac1(x, y) (x * y)
#define mac2(x, y) ((x) * (y))

void foo(void){
    int r;

    r = mac1(1 + 2, 3 + 4);       /* Non-compliant */
    r = mac1((1 + 2), (3 + 4));   /* Compliant */

    r = mac2(1 + 2, 3 + 4);       /* Compliant */
}
```

In this example, `mac1` and `mac2` are two defined macro expressions. The definition of `mac1` does not enclose the arguments in parentheses. In line 7, the macro expands to `r = (1 + 2 * 3 + 4);` This expression can be `(1 + (2 * 3) + 4)` or `(1 + 2) * (3 + 4)`. However, without parentheses, the program does not know the intended expression. Line 8 uses parentheses, so the line expands to `(1 + 2) * (3 + 4)`. This macro expression is compliant.

The definition of `mac2` does enclose the argument in parentheses. Line 10 (the same macro arguments in line 7) expands to `(1 + 2) * (3 + 4)`. This macro and macro expression are compliant.

Check Information

Group: Rec. 01. Preprocessor (PRE)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

PRE01-C

Introduced in R2019a

CERT C: Rec. PRE06-C

Enclose header files in an inclusion guard

Description

Rule Definition

Enclose header files in an inclusion guard.

Polyspace Implementation

This checker checks for **Contents of header file not guarded from multiple inclusions**.

Examples

Contents of header file not guarded from multiple inclusions

Issue

The issue occurs when you do not take precautions order to prevent the contents of a header file being included more than once.

If you include a header file whose contents are not guarded from multiple inclusion, the analysis raises a violation of this directive. The violation is shown at the beginning of the header file.

You can guard the contents of a header file from multiple inclusion by using one of the following methods:

```
<start-of-file>
#ifndef <control macro>
#define <control macro>
    /* Contents of file */
#endif
<end-of-file>
```

or

```
<start-of-file>
#ifdef <control macro>
#error ...
#else
#define <control macro>
    /* Contents of file */
#endif
<end-of-file>
```

Unless you use one of these methods, Polyspace flags the header file inclusion as noncompliant.

Risk

When a translation unit contains a complex hierarchy of nested header files, it is possible for a particular header file to be included more than once, leading to confusion. If this multiple inclusion

produces multiple or conflicting definitions, then your program can have undefined or erroneous behavior.

For instance, suppose that a header file contains:

```
#ifdef _WIN64
   int env_var;
#elseif
   long int env_var;
#endif
```

If the header file is contained in two inclusion paths, one that defines the macro `_WIN64` and another that undefines it, you can have conflicting definitions of `env_var`.

Example - Code After Macro Guard

```
#ifndef  __MY_MACRO__
#define __MY_MACRO__
    void func(void);
#endif
void func2(void);
```

If a header file contains this code, it is noncompliant because the macro guard does not cover the entire content of the header file. The line `void func2(void)` is outside the guard.

Note You can have comments outside the macro guard.

Example - Code Before Macro Guard

```
void func(void);
#ifndef  __MY_MACRO__
#define __MY_MACRO__
    void func2(void);
#endif
```

If a header file contains this code, it is noncompliant because the macro guard does not cover the entire content of the header file. The line `void func(void)` is outside the guard.

Note You can have comments outside the macro guard.

Example - Mismatch in Macro Guard

```
#ifndef  __MY_MACRO__
#define __MY_MARCO__
    void func(void);
    void func2(void);
#endif
```

If a header file contains this code, it is noncompliant because the macro name in the `#ifndef` statement is different from the name in the following `#define` statement.

Check Information

Group: Rec. 01. Preprocessor (PRE)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

PRE06-C

Introduced in R2019a

CERT C: Rec. PRE07-C

Avoid using repeated question marks

Description

Rule Definition

Avoid using repeated question marks.

Polyspace Implementation

This checker checks for **Use of trigraphs**.

Examples

Use of trigraphs

Issue

The issue occurs when you use trigraphs in your code.

The Polyspace analysis converts trigraphs to the equivalent character for the defect analysis. However, Polyspace also raises a MISRA violation.

The standard requires that trigraphs must be transformed *before* comments are removed during preprocessing. Therefore, Polyspace raises a violation of this rule even if a trigraph appears in code comments.

Risk

You denote trigraphs with two question marks followed by a specific third character (for instance,`'??-'` represents a `'~'` (tilde) character and `'??)'` represents a `']'`). These trigraphs can cause accidental confusion with other uses of two question marks.

Note Digraphs (`<: :>`, `<% %>`, `%:`, `%:%:`) are permitted because they are tokens.

Check Information

Group: Rec. 01. Preprocessor (PRE)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

PRE07-C

Introduced in R2019a

CERT C: Rec. PRE09-C

Do not replace secure functions with deprecated or obsolescent functions

Description

Rule Definition

Do not replace secure functions with deprecated or obsolescent functions.

Polyspace Implementation

This checker checks for **Use of dangerous standard function**.

Examples

Use of dangerous standard function

Issue

The **Use of dangerous standard function** check highlights uses of functions that are inherently dangerous or potentially dangerous given certain circumstances. The following table lists possibly dangerous functions, the risks of using each function, and what function to use instead.

Dangerous Function	Risk Level	Safer Function
`gets`	Inherently dangerous — You cannot control the length of input from the console.	`fgets`
`cin`	Inherently dangerous — You cannot control the length of input from the console.	Avoid or prefaces calls to `cin` with `cin.width`.
`strcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`strncpy`
`stpcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`stpncpy`
`lstrcpy` or `StrCpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy`
`strcat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`strncat`, `strlcat`, or `strcat_s`
`lstrcat` or `StrCat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`StringCbCat`, `StringCchCat`, `strncay`, `strcat_s`, or `strlcat`

Dangerous Function	Risk Level	Safer Function
`wcpcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`wcpncpy`
`wcscat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`wcsncat`, `wcslcat`, or `wcncat_s`
`wcscpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`wcsncpy`
`sprintf`	Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur.	`snprintf`
`vsprintf`	Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur.	`vsnprintf`

Risk

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Using `sprintf`

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (sprintf(dst, "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }
```

```
    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

Correction — Use `snprintf` with Buffer Size

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

Check Information

Group: Rec. 01. Preprocessor (PRE)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

PRE09-C

Introduced in R2019a

CERT C: Rec. PRE10-C

Wrap multistatement macros in a do-while loop

Description

Rule Definition

Wrap multistatement macros in a do-while loop.

Polyspace Implementation

This checker checks for **Macro with multiple statements**.

Examples

Macro with multiple statements

Issue

Macro with multiple statements occurs when a macro contains multiple semicolon-terminated statements, irrespective of whether the statements are enclosed in braces.

Risk

The macro expansion, in certain contexts such as an `if` condition or a loop, can lead to unintended program logic.

For instance, consider the macro:

```
#define RESET(x,y) \
   x=0; \
   y=0;
```

In an `if` statement such as:

```
if(checkSomeCondition)
   RESET(x,y);
```

the macro expands to:

```
if(checkSomething)
   x=0;
y=0;
```

which might be unexpected if you want both statements to be executed in an `if` block.

Fix

In a macro definition, wrap multiple statements in a `do...while(0)` loop.

For instance, in the preceding example, use the definition:

```
#define RESET(x,y) \
   do { \
```

```
     x=0; \
     y=0; \
   } while(0)
```

This macro is appropriate to expand in all contexts. The `while(0)` ensures that the statements are executed only once.

Alternatively, use inline functions in preference to function-like macros that involve multiple statements.

Note that the loop is required for the correct solution and wrapping the statements in braces alone does not fix the issue. The macro expansion can still lead to unintended code.

Example – Macro with Multiple Statements

```
#define RESET(x,y) \
   x=0; \
   y=0;

void func(int *x, int *y, int resetFlag){
    if(resetFlag)
        RESET(x,y);
}
```

In this example, the defect occurs because the macro `RESET` consists of multiple statements.

Correction – Wrap Multiple Statements of Macro in `do-while` Loop

Wrap the statements of the macro in a `do..while(0)` loop in the macro definition.

```
#define RESET(x,y) \
   do { \
     x=0; \
     y=0; \
   } while(0)

void func(int *x, int *y, int resetFlag){
    if(resetFlag)
        RESET(x,y);
}
```

Check Information

Group: Rec. 01. Preprocessor (PRE)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

PRE10-C

Introduced in R2020a

CERT C: Rec. PRE11-C

Do not conclude macro definitions with a semicolon

Description

Rule Definition

Do not conclude macro definitions with a semicolon.

Polyspace Implementation

This checker checks for **Macro terminated with a semicolon**.

Examples

Macro terminated with a semicolon

Issue

Macro terminated with a semicolon occurs when a macro that is invoked at least once has a definition ending with a semicolon.

Risk

If a macro definition ends with a semicolon, the macro expansion can lead to unintended program logic in certain contexts, such as within an expression.

For instance, consider the macro:

```
#define INC_BY_ONE(x) ++x;
```

If used in the expression:

```
res = INC_BY_ONE(x)%2;
```

the expression resolves to:

```
res = ++x; %2;
```

The value of `x+1` is assigned to `res`, which is probably unintended. The leftover standalone statement `%2;` is valid C code and can only be detected by enabling strict compiler warnings.

Fix

Do not end macro definitions with a semicolon. Leave it up to users of the macro to add a semicolon after the macro when needed.

Alternatively, use inline functions in preference to function-like macros that involve statements ending with semicolon.

Example – Spurious Semicolon in Macro Definition

```
#define WHILE_LOOP(n) while(n>0);
```

```
void performAction(int timeStep);

void main() {
    int loopIter = 100;
    WHILE_LOOP(loopIter) {
        performAction(loopIter);
        loopIter--;
    }
}
```

In this example, the defect occurs because the definition of the macro `WHILE_LOOP(n)` ends with a semicolon. As a result of the semicolon, the `while` loop has an empty body and the following statements run only once. It was probably intended that the loop must run 100 times.

Correction – Remove Semicolon from Macro Definition

Remove the trailing semicolon from the macro definition. Users of the macro can add a semicolon after the macro when needed. In this example, a semicolon is not required.

```
#define WHILE_LOOP(n) while(n>0)

void performAction(int timeStep);

void main() {
    int loopIter = 100;
    WHILE_LOOP(loopIter) {
        performAction(loopIter);
        loopIter--;
    }
}
```

Check Information

Group: Rec. 01. Preprocessor (PRE)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

PRE11-C

Introduced in R2020a

CERT C: Rec. DCL01-C

Do not reuse variable names in subscopes

Description

Rule Definition

Do not reuse variable names in subscopes.

Polyspace Implementation

This checker checks for **Variable shadowing**.

Examples

Variable shadowing

Issue

Variable shadowing occurs when a variable hides another variable of the same name in an outer scope.

For instance, if a local variable has the same name as a global variable, the local variable hides the global variable during its lifetime.

Risk

When two variables with the same name exist in an inner and outer scope, any reference to the variable name uses the variable in the inner scope. However, a developer or reviewer might incorrectly expect that the variable in the outer scope was used.

Fix

The fix depends on the root cause of the defect. For instance, suppose you refactor a function such that you use a local static variable in place of a global variable. In this case, the global variable is redundant and you can remove its declaration. Alternatively, if you are not sure if the global variable is used elsewhere, you can modify the name of the local static variable and all references within the function.

If the shadowing is intended and you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Variable Shadowing Error

```
#include <stdio.h>

int fact[5]={1,2,6,24,120};

int factorial(int n)
 {
  int fact=1;
  /*Defect: Local variable hides global array with same name */
```

```
  for(int i=1;i<=n;i++)
    fact*=i;

  return(fact);
 }
```

Inside the `factorial` function, the integer variable `fact` hides the global integer array `fact`.

Correction — Change Variable Name

One possible correction is to change the name of one of the variables, preferably the one with more local scope.

```
#include <stdio.h>

int fact[5]={1,2,6,24,120};

int factorial(int n)
 {
  /* Fix: Change name of local variable */
  int f=1;

  for(int i=1;i<=n;i++)
    f*=i;

  return(f);
 }
```

Check Information

Group: Rec. 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL01-C

Introduced in R2019a

CERT C: Rec. DCL02-C

Use visually distinct identifiers

Description

Rule Definition

Use visually distinct identifiers.

Polyspace Implementation

This checker checks for **Use of typographically ambiguous identifiers**.

Examples

Use of typographically ambiguous identifiers

Issue

The issue occurs when you use identifiers in the same name space with overlapping visibility and the identifiers are not typographically unambiguous.

Risk

What “unambiguous” means depends on the alphabet and language in which source code is written. When you use identifiers that are typographically close, you can confuse between them.

For the Latin alphabet as used in English words, at a minimum, the identifiers should not differ by:

- The interchange of a lowercase letter with its uppercase equivalent.
- The presence or absence of the underscore character.
- The interchange of the letter `O` and the digit `0`.
- The interchange of the letter `I` and the digit `1`.
- The interchange of the letter `I` and the letter `l`.
- The interchange of the letter `S` and the digit `5`.
- The interchange of the letter `Z` and the digit `2`.
- The interchange of the letter `n` and the letter `h`.
- The interchange of the letter `B` and the digit `8`.
- The interchange of the letters `rn` and the letter `m`.

Example - Typographically Ambiguous Identifiers

```
void func(void) {
    int id1_numval;
    int id1_num_val;  /* Non-compliant */

    int id2_numval;
    int id2_numVal;   /* Non-compliant */
```

```
    int id3_lvalue;
    int id3_Ivalue;   /* Non-compliant */

    int id4_xyz;
    int id4_xy2;      /* Non-compliant */

    int id5_zer0;
    int id5_zer0;     /* Non-compliant */

    int id6_rn;
    int id6_m;        /* Non-compliant */
}
```

In this example, the rule is violated when identifiers that can be confused for each other are used.

Check Information

Group: Rec. 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL02-C

Introduced in R2019a

CERT C: Rec. DCL06-C

Use meaningful symbolic constants to represent literal values

Description

Rule Definition

Use meaningful symbolic constants to represent literal values.

Polyspace Implementation

This checker checks for these issues:

- **Hard-coded buffer size**.
- **Hard-coded loop boundary**.

Examples

Hard-coded buffer size

Issue

Hard-coded buffer size occurs when you use a numerical value instead of a symbolic constant when declaring a memory buffer such as an array.

Risk

Hard-coded buffer size causes the following issues:

- Hard-coded buffer size increases the likelihood of mistakes and therefore maintenance costs. If a policy change requires developers to change the buffer size, they must change every occurrence of the buffer size in the code.
- Hard-constant constants can be exposed to attack if the code is disclosed.

Fix

Use a symbolic name instead of a hard-coded constant for buffer size. Symbolic names include `const`-qualified variables, `enum` constants, or macros.

`enum` constants are recommended.

- Macros are replaced by their constant values after preprocessing. Therefore, they can expose the loop boundary.
- `enum` constants are known at compilation time. Therefore, compilers can optimize the loops more efficiently.

  `const`-qualified variables are usually known at run time.

Example - Hard-Coded Buffer Size

```
int table[100];
```

```
void read(int);

void func(void) {
    for (int i=0; i<100; i++)
        read(table[i]);
}
```

In this example, the size of the array `table` is hard-coded.

Correction — Use Symbolic Name

One possible correction is to replace the hard-coded size with a symbolic name.

```
const int MAX_1 = 100;
#define MAX_2 100
enum { MAX_3 = 100 };

int table_1[MAX_1];
int table_2[MAX_2];
int table_3[MAX_3];

void read(int);

void func(void) {
    for (int i=0; i < MAX_1; i++)
        read(table_1[i]);
    for (int i=0; i < MAX_2; i++)
        read(table_2[i]);
    for (int i=0; i < MAX_3; i++)
        read(table_3[i]);
}
```

Hard-coded loop boundary

Issue

Hard-coded loop boundary occurs when you use a numerical value instead of symbolic constant for the boundary of a `for`, `while` or `do-while` loop.

Risk

Hard-coded loop boundary causes the following issues:

- Hard-coded loop boundary makes the code vulnerable to denial of service attacks when the loop involves time-consuming computation or resource allocation.
- Hard-coded loop boundary increases the likelihood of mistakes and maintenance costs. If a policy change requires developers to change the loop boundary, they must change every occurrence of the boundary in the code.

  For instance, the loop boundary is 10000 and represents the maximum number of client connections supported in a network server application. If the server supports more clients, you must change all instances of the loop boundary in your code. Even if the loop boundary occurs once, you have to search for a numerical value of `10000` in your code. The numerical value can occur in places other than the loop boundary. You must browse through those places before you find the loop boundary.

Fix

Use a symbolic name instead of a hard-coded constant for loop boundary. Symbolic names include `const`-qualified variables, `enum` constants or macros.`enum` constants are recommended because:

- Macros are replaced by their constant values after preprocessing. Therefore, they can expose the buffer size.
- `enum` constants are known at compilation time. Therefore, compilers can allocate storage for them more efficiently.

  `const`-qualified variables are usually known at run time.

Example - Hard-Coded Loop Boundary

```
void performOperation(int);

void func(void) {
    for (int i=0; i<100; i++)
        performOperation(i);
}
```

In this example, the boundary of the `for` loop is hard-coded.

Correction — Use Symbolic Name

One possible correction is to replace the hard-coded loop boundary with a symbolic name.

```
const int MAX_1 = 100;
#define MAX_2 100
enum { MAX_3 = 100 };

void performOperation_1(int);
void performOperation_2(int);
void performOperation_3(int);

void func(void) {
    for (int i=0; i<MAX_1; i++)
        performOperation_1(i);
    for (int i=0; i<MAX_2; i++)
        performOperation_2(i);
    for (int i=0; i<MAX_3; i++)
        performOperation_3(i);
}
```

Check Information

Group: Rec. 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL06-C

Introduced in R2019a

CERT C: Rec. DCL07-C

Include the appropriate type information in function declarators

Description

Rule Definition

Include the appropriate type information in function declarators.

Polyspace Implementation

This checker checks for these issues:

- **Cast between function pointers with different types**.
- **Function declared implicitly**.

Examples

Cast between function pointers with different types

Issue

The issues occurs when you perform a conversion between a pointer to a function and any other type.

Polyspace considers both explicit and implicit casts when checking this rule. However, casts from `NULL` or `(void*)0` do not violate this rule.

Risk

The rule forbids the following two conversions:

- Conversion from a function pointer to any other type. This conversion causes undefined behavior.
- Conversion from a function pointer to another function pointer, if the function pointers have different argument and return types.

  The conversion is forbidden because calling a function through a pointer with incompatible type results in undefined behavior.

Example - Cast between two function pointers

```
typedef void (*fp16) (short n);
typedef void (*fp32) (int n);

#include <stdlib.h>                     /* To obtain macro  NULL */

void func(void) {   /* Exception 1 - Can convert a null pointer
                     * constant into a pointer to a function */
  fp16 fp1 = NULL;                 /* Compliant - exception  */
  fp16 fp2 = (fp16) fp1;           /* Compliant */
  fp32 fp3 = (fp32) fp1;           /* Non-compliant */
  if (fp2 != NULL) {}              /* Compliant - exception  */
  fp16 fp4 = (fp16) 0x8000;        /* Non-compliant - integer to
                                    * function pointer */}
```

In this example, the rule is violated when:

- The pointer `fp1` of type `fp16` is cast to type `fp32`. The function pointer types `fp16` and `fp32` have different argument types.
- An integer is cast to type `fp16`.

The rule is not violated when function pointers `fp1` and `fp2` are cast to `NULL`.

Function declared implicitly

Issue

The issue occurs when you declare a function implicitly.

Risk

An implicit declaration occurs when you call a function before declaring or defining it. When you declare a function explicitly before calling it, the compiler can match the argument and return types with the parameter types in the declaration. If an implicit declaration occurs, the compiler makes assumptions about the argument and return types. For instance, it assumes a return type of `int`. The assumptions might not agree with what you expect and cause undesired type conversions.

Example - Function Not Declared Before Call

```
#include <math.h>

extern double power3 (double val, int exponent);
int getChoice(void);

double func() {
    double res;
    int ch = getChoice();
    if(ch == 0) {
        res = power(2.0, 10);    /* Non-compliant */
    }
    else if( ch==1) {
        res = power2(2.0, 10);   /* Non-compliant */
    }
    else {
        res = power3(2.0, 10);   /* Compliant */
        return res;
    }
}

double power2 (double val, int exponent) {
    return (pow(val, exponent));
}
```

In this example, the rule is violated when a function that is not declared is called in the code. Even if a function definition exists later in the code, the rule violation occurs.

The rule is not violated when the function is declared before it is called in the code. If the function definition exists in another file and is available only during the link phase, you can declare the function in one of the following ways:

- Declare the function with the `extern` keyword in the current file.
- Declare the function in a header file and include the header file in the current file.

Check Information

Group: Rec. 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL07-C

Introduced in R2019a

CERT C: Rec. DCL10-C

Maintain the contract between the writer and caller of variadic functions

Description

Rule Definition

Maintain the contract between the writer and caller of variadic functions.

Polyspace Implementation

This checker checks for **Format string specifiers and arguments mismatch**.

Examples

Format string specifiers and arguments mismatch

Issue

Format string specifiers and arguments mismatch occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
  const char *message = "License not available";
  int err_number = ;-4
  printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Printing a Float

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", fst);
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

Check Information

Group: Rec. 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL10-C

Introduced in R2019a

CERT C: Rec. DCL11-C

Understand the type issues associated with variadic functions

Description

Rule Definition

Understand the type issues associated with variadic functions.

Polyspace Implementation

This checker checks for **Format string specifiers and arguments mismatch**.

Examples

Format string specifiers and arguments mismatch

Issue

Format string specifiers and arguments mismatch occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
  const char *message = "License not available";
  int err_number = ;-4
  printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Printing a Float

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", fst);
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

Check Information

Group: Rec. 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL11-C

Introduced in R2019a

CERT C: Rec. DCL12-C

Implement abstract data types using opaque types

Description

Rule Definition

Implement abstract data types using opaque types.

Polyspace Implementation

This checker checks for **Structure or union object implementation visible in file where pointer to this object is not dereferenced**.

Examples

Structure or union object implementation visible in file where pointer to this object is not dereferenced

Issue

The issue occurs when a pointer to a structure or union is never dereferenced within a translation unit, but the implementation of the object is not hidden.

If a structure or union is defined in a file or a header file included in the file, a pointer to this structure or union declared but the pointer never dereferenced in the file, the checker flags a coding rule violation. The structure or union definition should not be visible to this file.

If you see a violation of this rule on a structure definition, identify if you have defined a pointer to the structure in the same file or in a header file included in the file. Then check if you dereference the pointer anywhere in the file. If you do not dereference the pointer, the structure definition should be hidden from this file and included header files.

`file.h`: Contains structure implementation.

```
#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct  {
  int a;
} myStruct;

#endif
```

`file.c`: Includes `file.h` but does not dereference structure.

```
#include "file.h"

myStruct* getObj(void);
void useObj(myStruct*);

void func() {
```

```
  myStruct *sPtr = getObj();
  useObj(sPtr);
}
```

In this example, the pointer to the type `myStruct` is not dereferenced. The pointer is simply obtained from the `getObj` function and passed to the `useObj` function.

The implementation of `myStruct` is visible in the translation unit consisting of `file.c` and `file.h`.

One possible correction is to define an opaque data type in the header file `file.h`. The opaque data type `ptrMyStruct` points to the `myStruct` structure without revealing what the structure contains. The structure `myStruct` itself can be defined in a separate translation unit, in this case, consisting of the file `file2.c`. The common header file `file.h` must be included in both `file.c` and `file2.c` for linking the structure definition to the opaque type definition.

`file.h`: Does not contain structure implementation.

```
#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct myStruct *ptrMyStruct;

ptrMyStruct getObj(void);
void useObj(ptrMyStruct);

#endif
```

`file.c`: Includes `file.h` but does not dereference structure.

```
#include "file.h"

void func() {
  ptrMyStruct sPtr = getObj();
  useObj(sPtr);
}
```

`file2.c`: Includes `file.h` and dereferences structure.

```
#include "file.h"

struct myStruct {
  int a;
};

void useObj(ptrMyStruct ptr) {
    (ptr->a)++;
}
```

Risk

If a pointer to a structure or union is not dereferenced in a file, the implementation details of the structure or union need not be available in the translation unit for the file. You can hide the implementation details such as structure members and protect them from unintentional changes.

Define an opaque type that can be referenced via pointers but whose contents cannot be accessed.

Example - Object Implementation Revealed

`file.h`: Contains structure implementation.

```
#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct  {
  int a;
} myStruct;

#endif
```

`file.c`: Includes `file.h` but does not dereference structure.

```
#include "file.h"

myStruct* getObj(void);
void useObj(myStruct*);

void func() {
  myStruct *sPtr = getObj();
  useObj(sPtr);
}
```

In this example, the pointer to the type `myStruct` is not dereferenced. The pointer is simply obtained from the `getObj` function and passed to the `useObj` function.

The implementation of `myStruct` is visible in the translation unit consisting of `file.c` and `file.h`.

Correction — Define Opaque Type

One possible correction is to define an opaque data type in the header file `file.h`. The opaque data type `ptrMyStruct` points to the `myStruct` structure without revealing what the structure contains. The structure `myStruct` itself can be defined in a separate translation unit, in this case, consisting of the file `file2.c`. The common header file `file.h` must be included in both `file.c` and `file2.c` for linking the structure definition to the opaque type definition.

`file.h`: Does not contain structure implementation.

```
#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct myStruct *ptrMyStruct;

ptrMyStruct getObj(void);
void useObj(ptrMyStruct);

#endif
```

`file.c`: Includes `file.h` but does not dereference structure.

```
#include "file.h"

void func() {
  ptrMyStruct sPtr = getObj();
  useObj(sPtr);
}
```

`file2.c`: Includes `file.h` and dereferences structure.

```
#include "file.h"
```

```
struct myStruct {
  int a;
};

void useObj(ptrMyStruct ptr) {
    (ptr->a)++;
}
```

Check Information

Group: Rec. 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL12-C

Introduced in R2019a

CERT C: Rec. DCL13-C

Declare function parameters that are pointers to values not changed by the function as const

Description

Rule Definition

Declare function parameters that are pointers to values not changed by the function as const.

Polyspace Implementation

This checker checks for **Pointer to non-cont qualified function parameter**.

Examples

Pointer to non-`cont` qualified function parameter

Issue

The rule checker flags a pointer to a non-`const` function parameter if the pointer does not modify the addressed object. The assumption is that the pointer is not meant to modify the object and so must point to a `const`-qualified type.

Risk

This rule ensures that you do not inadvertently use pointers to modify objects.

Example - Pointer That Should Point to `const`-Qualified Types

```
#include <string.h>

typedef unsigned short uint16_t;

uint16_t ptr_ex(uint16_t *p) {      /* Non-compliant */
    return *p;
}

char last_char(char * const s){     /* Non-compliant */
    return s[strlen(s) - 1u];
}

uint16_t first(uint16_t a[5]){      /* Non-compliant */
    return a[0];
}
```

This example shows three different noncompliant pointer parameters.

- In the `ptr_ex` function, `p` does not modify an object. However, the type to which `p` points is not `const`-qualified, so it is noncompliant.
- In `last_char`, the pointer `s` is `const`-qualified but the type it points to is not. This parameter is noncompliant because `s` does not modify an object.
- The function `first` does not modify the elements of the array `a`. However, the element type is not `const`-qualified, so `a` is also noncompliant.

Correction — Use `const` Keywords

One possible correction is to add `const` qualifiers to the definitions.

```
#include <string.h>

typedef unsigned short uint16_t;

uint16_t ptr_ex(const uint16_t *p){     /* Compliant */
    return *p;
}

char last_char(const char * const s){   /* Compliant */
    return s[strlen( s ) - 1u];
}

uint16_t first(const uint16_t a[5]) {   /* Compliant */
    return a[0];
}
```

Check Information

Group: Rec. 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL13-C

Introduced in R2019a

CERT C: Rec. DCL15-C

Declare file-scope objects or functions that do not need external linkage as static

Description

Rule Definition

Declare file-scope objects or functions that do not need external linkage as static.

Polyspace Implementation

This checker checks for **Function or object with external linkage referenced in only one translation unit**.

Examples

Function or object with external linkage referenced in only one translation unit

Issue

The rule checker flags:

- Objects that are defined at file scope without the `static` specifier but used only in one file.
- Functions that are defined without the `static` specifier but called only in one file.

If you intend to use the object or function in one file only, declare it static.

Objects that are defined at file scope without the `static` specifier but used only in one file.

Functions that are defined without the `static` specifier but called only in one file.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Risk

Compliance with this rule avoids confusion between your identifier and an identical identifier in another translation unit or library. If you restrict or reduce the visibility of an object by giving it internal linkage or no linkage, you or someone else is less likely to access the object inadvertently.

Example - Variable with External Linkage Used in One File

Header file:

```
/* file.h */
extern int var;
```

First source file:

```
/* file1.c */
#include "file.h"

int var;    /* Compliant */
```

```
int var2;   /* Non compliant */
static int var3; /* Compliant */

void reset(void);

void reset(void) {
    var = 0;
    var2 = 0;
    var3 = 0;
}
```

Second source file:

```
/* file2.c */
#include "file.h"

void increment(int var2);

void increment(int var2) {
    var++;
    var2++;
}
```

In this example:

- The declaration of `var` is compliant because `var` is declared with external linkage and used in multiple files.
- The declaration of `var2` is noncompliant because `var2` is declared with external linkage but used in one file only.

  It might appear that `var2` is defined in both files. However, in the second file, `var2` is a parameter with no linkage and is not the same as the `var2` in the first file.
- The declaration of `var3` is compliant because `var3` is declared with internal linkage (with the `static` specifier) and used in one file only.

Example - Function with External Linkage Used in One File

Header file:

```
/* file.h */
extern int var;
extern void increment1 (void);
```

First source file:

```
/* file1.c */
#include "file.h"

int var;

void increment2(void);
static void increment3(void);
void func(void);

void increment2(void) { /* Non compliant */
    var+=2;
}
```

```
static void increment3(void) { /* Compliant */
    var+=3;
}

void func(void) {
    increment1();
    increment2();
    increment3();
}
```

Second source file:

```
/* file2.c */
#include "file.h"

void increment1(void) { /* Compliant */
    var++;
}
```

In this example:

- The definition of `increment1` is compliant because `increment1` is defined with external linkage and called in a different file.
- The declaration of `increment2` is noncompliant because `increment2` is defined with external linkage but called in the same file and nowhere else.
- The declaration of `increment3` is compliant because `increment3` is defined with internal linkage (with the `static` specifier) and called in the same file and nowhere else.

Check Information

Group: Rec. 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL15-C

Introduced in R2019a

CERT C: Rec. DCL16-C

Use 'L,' not 'l,' to indicate a long value

Description

Rule Definition

Use 'L,' not 'l,' to indicate a long value.

Polyspace Implementation

This checker checks for **Use of lowercase "l" in literal suffix**.

Examples

Use of lowercase "l" in literal suffix

Issue

The issue occurs when you use the lowercase character “l” in a literal suffix.

Risk

The lowercase character “l” can be confused with the digit “1”. Use the uppercase “L” instead.

Check Information

Group: Rec. 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL16-C

Introduced in R2019a

CERT C: Rec. DCL18-C

Do not begin integer constants with 0 when specifying a decimal value

Description

Rule Definition

Do not begin integer constants with 0 when specifying a decimal value.

Polyspace Implementation

This checker checks for **Use of octal constants**.

Examples

Use of octal constants

Issue

If you use octal constants in a macro definition, the rule checker flags the issue even if the macro is not used.

Risk

Octal constants are denoted by a leading zero. Developers can mistake an octal constant as a decimal constant with a redundant leading zero.

Example - Use of octal constants

```
#define CST     021
#define VALUE   010             /* Compliant - constant not used */
#if 010 == 01                   /* Non-Compliant - constant used */
#define CST 021                 /* Non-Compliant - constant not used */
#endif

extern short code[5];
static char* str2 = "abcd\0efg";  /* Compliant */

void main(void) {
    int value1 = 0;              /* Compliant */
    int value2 = 01;             /* Non-Compliant - decimal 01 */
    int value3 = 1;              /* Compliant */
    int value4 = '\109';         /* Compliant */

    code[1] = 109;               /* Compliant     - decimal 109 */
    code[2] = 100;               /* Compliant     - decimal 100 */
    code[3] = 052;               /* Non-Compliant - decimal 42 */
    code[4] = 071;               /* Non-Compliant - decimal 57 */

    if (value1 != CST) {         /* Non-Compliant - decimal 17 */
        value1 = !(value1 != 0);  /* Compliant */
    }
}
```

In this example, the rule is not violated when octal constants are used to define macros `CST` and `VALUE`. The rule is violated only when the macros are used.

Check Information

Group: Rec. 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL18-C

Introduced in R2019a

CERT C: Rec. DCL19-C

Minimize the scope of variables and functions

Description

Rule Definition

Minimize the scope of variables and functions.

Polyspace Implementation

This checker checks for these issues:

- **Function or object declared without static specifier and referenced in only one file**.
- **Object defined beyond necessary scope**.

Examples

Function or object declared without `static` specifier and referenced in only one file

Issue

The rule checker flags:

- Objects that are defined at file scope without the `static` specifier but used only in one file.
- Functions that are defined without the `static` specifier but called only in one file.

If you intend to use the object or function in one file only, declare it static.

Objects that are defined at file scope without the `static` specifier but used only in one file.

Functions that are defined without the `static` specifier but called only in one file.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Risk

Compliance with this rule avoids confusion between your identifier and an identical identifier in another translation unit or library. If you restrict or reduce the visibility of an object by giving it internal linkage or no linkage, you or someone else is less likely to access the object inadvertently.

Example - Variable with External Linkage Used in One File

Header file:

```
/* file.h */
extern int var;
```

First source file:

```
/* file1.c */
#include "file.h"
```

```
int var;    /* Compliant */
int var2;   /* Non compliant */
static int var3; /* Compliant */

void reset(void);

void reset(void) {
    var = 0;
    var2 = 0;
    var3 = 0;
}
```

Second source file:

```
/* file2.c */
#include "file.h"

void increment(int var2);

void increment(int var2) {
    var++;
    var2++;
}
```

In this example:

- The declaration of `var` is compliant because `var` is declared with external linkage and used in multiple files.
- The declaration of `var2` is noncompliant because `var2` is declared with external linkage but used in one file only.

  It might appear that `var2` is defined in both files. However, in the second file, `var2` is a parameter with no linkage and is not the same as the `var2` in the first file.
- The declaration of `var3` is compliant because `var3` is declared with internal linkage (with the `static` specifier) and used in one file only.

Example - Function with External Linkage Used in One File

Header file:

```
/* file.h */
extern int var;
extern void increment1 (void);
```

First source file:

```
/* file1.c */
#include "file.h"

int var;

void increment2(void);
static void increment3(void);
void func(void);

void increment2(void) { /* Non compliant */
```

```
    var+=2;
}

static void increment3(void) { /* Compliant */
    var+=3;
}

void func(void) {
    increment1();
    increment2();
    increment3();
}
```

Second source file:

```
/* file2.c */
#include "file.h"

void increment1(void) { /* Compliant */
    var++;
}
```

In this example:

- The definition of `increment1` is compliant because `increment1` is defined with external linkage and called in a different file.
- The declaration of `increment2` is noncompliant because `increment2` is defined with external linkage but called in the same file and nowhere else.
- The declaration of `increment3` is compliant because `increment3` is defined with internal linkage (with the `static` specifier) and called in the same file and nowhere else.

Object defined beyond necessary scope

Issue

The issue occurs when the identifier of an object only appears in a single function but the object is defined beyond the block scope.

The rule checker flags `static` objects that are accessed in one function only but declared at file scope.

Risk

If you define an object at block scope, you or someone else is less likely to access the object inadvertently outside the block.

Example - Object Declared at File Scope but Used in One Function

```
static int ctr;   /* Non compliant */

int checkStatus(void);
void incrementCount(void);

void incrementCount(void) {
    ctr=0;
    while(1) {
        if(checkStatus())
            ctr++;
```

```
    }
}
```

In this example, the declaration of `ctr` is noncompliant because it is declared at file scope but used only in the function `incrementCount`. Declare `ctr` in the body of `incrementCount` to be MISRA C-compliant.

Check Information

Group: Rec. 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL19-C

Introduced in R2019a

CERT C: Rec. DCL22-C

Use volatile for data that cannot be cached

Description

Rule Definition

Use volatile for data that cannot be cached.

Polyspace Implementation

This checker checks for **Write without a further read**.

Examples

Write without a further read

Issue

Write without a further read occurs when a value assigned to a variable is never read.

For instance, you write a value to a variable and then write a second value before reading the previous value. The first write operation is redundant.

Risk

Redundant write operations often indicate programming errors. For instance, you forgot to read the variable between two successive write operations or unintentionally read a different variable.

Fix

Identify the reason why you write to the variable but do not read it later. Look for common programming errors such as accidentally reading a different variable with a similar name.

If you determine that the write operation is redundant, remove the operation.

Example - Write Without Further Read Error

```
void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();
    /* Defect: Useless write */
}
```

After the variable `level` gets assigned the value `4 * getsensor()`, it is not read.

Correction — Use Value After Assignment

One possible correction is to use the variable `level` after the assignment.

```
#include <stdio.h>

void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();

    /* Fix: Use level after assignment */
    printf("The value is %d", level);

}
```

The variable `level` is printed, reading the new value.

Check Information

Group: Rec. 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL22-C

Introduced in R2019a

CERT C: Rec. DCL23-C

Guarantee that mutually visible identifiers are unique

Description

Rule Definition

Guarantee that mutually visible identifiers are unique.

Polyspace Implementation

This checker checks for these issues:

- **External identifiers not distinct**.
- **Identifier in same scope and namespace not distinct**.
- **Macro identifier not distinct**.
- **Name for macros and identifiers not distinct**.

Examples

External identifiers not distinct

Issue

The issue occurs when external identifiers have the same first six characters for C90 or the same first 31 characters for C99.

Risk

External identifiers are ones declared with global scope or storage class `extern`.

Polyspace considers two names as distinct if there is a difference between their first 31 characters. If the difference between two names occurs only beyond the first 31 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first six characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`.

Example - C90: First Six Characters of Identifiers Not Unique

```
int engine_temperature_raw;
int engine_temperature_scaled;   /* Non-compliant */
int engin2_temperature;          /* Compliant */
```

In this example, the identifier `engine_temperature_scaled` has the same first six characters as a previous identifier, `engine_temperature_raw`.

Example - C99: First 31 Characters of Identifiers Not Unique

```
int engine_exhaust_gas_temperature_raw;
int engine_exhaust_gas_temperature_scaled; /* Non-compliant */

int eng_exhaust_gas_temp_raw;
int eng_exhaust_gas_temp_scaled;           /* Compliant */
```

In this example, the identifier `engine_exhaust_gas_temperature_scaled` has the same first 31 characters as a previous identifier, `engine_exhaust_gas_temperature_raw`.

Example - C90: First Six Characters Identifiers in Different Translation Units Differ in Case Alone

```
/* file1.c */
int abc = 0;

/* file2.c */
int ABC = 0; /* Non-compliant */
```

In this example, the implementation supports six significant case-insensitive characters in *external identifiers*. The identifiers in the two translations are different but are not distinct in their significant characters.

Identifier in same scope and namespace not distinct

Issue

The issue occurs when you declare identifiers in the same scope and namespace and the identifiers have the same first 31 characters in C90 or the same first 63 characters in C99.

Risk

Polyspace considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`.

Example - C90: First 31 Characters of Identifiers Not Unique

```
extern int engine_exhaust_gas_temperature_raw;
static int engine_exhaust_gas_temperature_scaled;      /* Non-compliant */

extern double engine_exhaust_gas_temperature_raw;
static double engine_exhaust_gas_temperature2_scaled;  /* Compliant */

void func ( void )
{
  /* Not in the same scope */
  int engine_exhaust_gas_temperature_local;            /* Compliant */
}
```

In this example, the identifier `engine_exhaust_gas_temperature_scaled` has the same 31 characters as a previous identifier, `engine_exhaust_gas_temperature_raw`.

The rule does not apply if the two identifiers have the same 31 characters but have different scopes. For instance, `engine_exhaust_gas_temperature_local` has the same 31 characters as `engine_exhaust_gas_temperature_raw` but different scope.

Example - C99: First 63 Characters of Identifiers Not Unique

```
extern int engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_raw;
static int engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_scale;
    /* Non-compliant */

extern int engine_gas_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx__raw;
static int engine_gas_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx__scale;
```

```
    /* Compliant */

void func ( void )
{
/* Not in the same scope */
    int engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_local;
          /* Compliant */
}
```

In this example, the identifier `engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_scale` has the same 63 characters as a previous identifier, `engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_raw`.

Macro identifier not distinct

Issue

The issue occurs when you use macro identifiers that have the same first 31 characters in C90 or the same first 63 characters in C99.

Risk

The names of macro identifiers must be distinct from both other macro identifiers and their parameters.

Polyspace considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`.

Example - C90: First 31 Characters of Macro Names Not Unique

```
#define engine_exhaust_gas_temperature_raw egt_r
#define engine_exhaust_gas_temperature_scaled egt_s   /* Non-compliant */

#define engine_exhaust_gas_temp_raw egt_r
#define engine_exhaust_gas_temp_scaled egt_s          /* Compliant */
```

In this example, the macro `engine_exhaust_gas_temperature_scaled egt_s` has the same first 31 characters as a previous macro `engine_exhaust_gas_temperature_scaled`.

Example - C99: First 63 Characters of Macro Names Not Unique

```
#define engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_raw egt_r
#define engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_raw_scaled egt_s
    /* Non-compliant */

/* 63 significant case-sensitive characters in macro identifiers */
#define new_engine_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_raw egt_r
#define new_engine_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_scaled egt_s
    /* Compliant */
```

In this example, the macro `engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx___gaz_scaled` has

the same first 63 characters as a previous macro `engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx___raw`.

Name for macros and identifiers not distinct

Issue

The issue occurs when identifiers are not distinct from macro names.

Risk

The rule requires that macro names that exist only prior to processing must be different from identifier names that also exist after preprocessing. Keeping macro names and identifiers distinct help avoid confusion.

Polyspace considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`.

Example - Macro Names Same as Identifier Names

```
#define Sum_1(x, y) ( ( x ) + ( y ) )
short Sum_1;                    /* Non-compliant */

#define Sum_2(x, y) ( ( x ) + ( y ) )
short x = Sum_2 ( 1, 2 );       /* Compliant */
```

In this example, `Sum_1` is both the name of an identifier and a macro. `Sum_2` is used only as a macro.

Example - C90: First 31 Characters of Macro Name Same as Identifier Name

```
#define       low_pressure_turbine_temperature_1 lp_tb_temp_1
static int low_pressure_turbine_temperature_2;      /* Non-compliant  */
```

In this example, the identifier `low_pressure_turbine_temperature_2` has the same first 31 characters as a previous macro `low_pressure_turbine_temperature_1`.

Check Information

Group: Rec. 02. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

DCL23-C

Introduced in R2019a

CERT C: Rec. EXP00-C

Use parentheses for precedence of operation

Description

Rule Definition

Use parentheses for precedence of operation.

Polyspace Implementation

This checker checks for **Possibly unintended evaluation of expression because of operator precedence rules**.

Examples

Possibly unintended evaluation of expression because of operator precedence rules

Issue

Possibly unintended evaluation of expression because of operator precedence rules occurs when an arithmetic expression result is possibly unintended because operator precedence rules dictate an evaluation order that you do not expect.

The defect highlights expressions of the form `x` *`op_1`* `y` *`op_2`* `z`. Here, *`op_1`* and *`op_2`* are operator combinations that commonly induce this error. For instance, `x == y | z`.

The checker does not flag all operator combinations. For instance, `x == y || z` is not flagged because you most likely intended to perform a logical OR between `x == y` and `z`. Specifically, the checker flags these combinations:

- && and ||: For instance, `x || y && z` or `x && y || z`.
- Assignment and bitwise operations: For instance, `x = y | z`.
- Assignment and comparison operations: For instance, `x = y != z` or `x = y > z`.
- Comparison operations: For instance, `x > y > z` (except when one of the comparisons is an equality `x == y > z`).
- Shift and numerical operation: For instance, `x << y + 2`.
- Pointer dereference and arithmetic: For instance, `*p++`.

Risk

The defect can cause the following issues:

- If you or another code reviewer reviews the code, the intended order of evaluation is not immediately clear.
- It is possible that the result of the evaluation does not meet your expectations. For instance:
  - In the operation `*p++`, it is possible that you expect the dereferenced value to be incremented. However, the pointer `p` is incremented before the dereference.

- In the operation `(x == y | z)`, it is possible that you expect `x` to be compared with `y | z`. However, the == operation happens before the | operation.

Fix

See if the order of evaluation is what you intend. If not, apply parentheses to implement the evaluation order that you want.

For better readability of your code, it is good practice to apply parenthesis to implement an evaluation order even when operator precedence rules impose that order.

Example - Expressions with Possibly Unintended Evaluation Order

```
int test(int a, int b, int c) {
    return(a & b == c);
}
```

In this example, the == operation happens first, followed by the & operation. If you intended the reverse order of operations, the result is not what you expect.

Correction — Parenthesis For Intended Order

One possible correction is to apply parenthesis to implement the intended evaluation order.

```
int test(int a, int b, int c) {
    return((a & b) == c);
}
```

Check Information

Group: Rec. 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP00-C

Introduced in R2019a

CERT C: Rec. EXP05-C

Do not cast away a const qualification

Description

Rule Definition

Do not cast away a const qualification.

Polyspace Implementation

This checker checks for **Cast to pointer that removes const qualification**.

Examples

Cast to pointer that removes `const` qualification

Issue

Polyspace flags both implicit and explicit conversions that violate this rule.

Risk

This rule forbids casts from a pointer to a `const` object to a pointer that does not point to a `const` object.

Such casts violate type qualification. For example, the `const` qualifier indicates the read-only status of an object. If a cast removes the qualifier, the object is no longer read-only.

Example - Casts That Remove Qualifiers

```
void foo(void) {

    /* Cast on simple type */
    unsigned short           x;
    unsigned short * const   cpi = &x;  /* const pointer */
    unsigned short * const  *pcpi;   /* pointer to const pointer */
    unsigned short **ppi;
    const unsigned short    *pci;    /* pointer to const */
    unsigned short          *pi;

    pi = cpi;                        /* Compliant - no cast required */
    pi  = (unsigned short *)  pci;   /* Non-compliant */
    ppi = (unsigned short **)pcpi;   /* Non-compliant */
}
```

In this example, the variables `pci` and `pcpi` have the `const` qualifier in their type. The rule is violated when the variables are cast to types that do not have the `const` qualifier.

Even though `cpi` has a `const` qualifier in its type, the rule is not violated in the statement `p=cpi;`. The assignment does not cause a type conversion because both `p` and `cpi` have type `unsigned short`.

Check Information

Group: Rec. 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP05-C

Introduced in R2019a

CERT C: Rec. EXP08-C

Ensure pointer arithmetic is used correctly

Description

Rule Definition

Ensure pointer arithmetic is used correctly.

Polyspace Implementation

This checker checks for these issues:

- **Array access out of bounds**
- **Pointer access out of bounds**.
- **Subtraction between pointers to different arrays**.
- **Incorrect pointer scaling**.

Examples

Array access out of bounds

Issue

This issue occurs when an array index falls outside the range `[0...array_size-1]` during array access.

Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example – Array Access Out of Bounds Error

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

Correction – Keep Array Index Within Array Bounds

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

Pointer access out of bounds

Issue

This issue occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example – Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

Subtraction between pointers to different arrays

Issue

This rule is raised whenever the analysis detects a `Subtraction or comparison between pointers to different arrays`.

Risk

This rule applies to expressions of the form `pointer_expression1 - pointer_expression2`. The behavior is undefined if `pointer_expression1` and `pointer_expression2`:

- Do not point to elements of the same array,
- Or do not point to the element one beyond the end of the array.

Example - Subtracting Pointers

```
#include <stddef.h>

void f1 (int32_t *ptr)
{
    int32_t a1[10];
    int32_t a2[10];
    int32_t *p1 = &a1[ 1];
    int32_t *p2 = &a2[10];
    ptrdiff_t diff1, diff2, diff3;

    diff1 =  p1 - a1;   // Compliant
    diff2 =  p2 - a2;   // Compliant
    diff3 =  p1 - p2;   // Non-compliant
}
```

In this example, the three subtraction expressions show the difference between compliant and noncompliant pointer subtractions. The `diff1` and `diff2` subtractions are compliant because the pointers point to the same array. The `diff3` subtraction is not compliant because `p1` and `p2` point to different arrays.

Incorrect pointer scaling

Issue

Incorrect pointer scaling occurs when Polyspace Bug Finder considers that you are ignoring the implicit scaling in pointer arithmetic.

For instance, the defect can occur in the following situations.

Situation	Risk	Possible Fix
You use the `sizeof` operator in arithmetic operations on a pointer.	The `sizeof` operator returns the size of a data type in number of bytes. Pointer arithmetic is already implicitly scaled by the size of the data type of the pointed variable. Therefore, the use of `sizeof` in pointer arithmetic produces unintended results.	Do not use `sizeof` operator in pointer arithmetic.
You perform arithmetic operations on a pointer, and then apply a cast.	Pointer arithmetic is implicitly scaled. If you do not consider this implicit scaling, casting the result of a pointer arithmetic produces unintended results.	Apply the cast before the pointer arithmetic.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Use of `sizeof` Operator

```
void func(void) {
    int arr[5] = {1,2,3,4,5};
    int *ptr = arr;

    int value_in_position_2 = *(ptr + 2*(sizeof(int)));
}
```

In this example, the operation `2*(sizeof(int))` returns twice the size of an `int` variable in bytes. However, because pointer arithmetic is implicitly scaled, the number of bytes by which `ptr` is offset is `2*(sizeof(int))*(sizeof(int))`.

In this example, the incorrect scaling shifts `ptr` outside the bounds of the array. Therefore, a **Pointer access out of bounds** error appears on the * operation.

Correction — Remove `sizeof` Operator

One possible correction is to remove the `sizeof` operator.

```
void func(void) {
    int arr[5] = {1,2,3,4,5};
    int *ptr = arr;

    int value_in_position_2 = *(ptr + 2);
}
```

Example - Cast Following Pointer Arithmetic

```
int func(void) {
    int x = 0;
    char r = *(char *)(&x + 1);
    return r;
}
```

In this example, the operation `&x + 1` offsets `&x` by `sizeof(int)`. Following the operation, the resulting pointer points outside the allowed buffer. When you dereference the pointer, a **Pointer access out of bounds** error appears on the * operation.

Correction — Apply Cast Before Pointer Arithmetic

If you want to access the second byte of `x`, first cast `&x` to a `char*` pointer and then perform the pointer arithmetic. The resulting pointer is offset by `sizeof(char)` bytes and still points within the allowed buffer, whose size is `sizeof(int)` bytes.

```
int func(void) {
    int x = 0;
    char r = *((char *)(&x )+ 1);
    return r;
}
```

Check Information

Group: Rec. 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP08-C

Introduced in R2019a

CERT C: Rec. EXP09-C

Use sizeof to determine the size of a type or variable

Description

Rule Definition

Use sizeof to determine the size of a type or variable.

Polyspace Implementation

This checker checks for **Hard-coded object size used to manipulate memory**.

Examples

Hard-coded object size used to manipulate memory

Issue

Hard-coded object size used to manipulate memory occurs on constants that are memory size arguments for memory functions such as `malloc` or `memset`.

Risk

If you hard code object size, your code is not portable to architectures with different type sizes. If the constant value is not the same as the object size, the buffer might or might not overflow.

Fix

For the size argument of memory functions, use `sizeof(`*`object`*`)`.

Example - Assume 4-Byte Integer Pointers

```
#include <stddef.h>
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};
extern void fill_ints(int **matrix, size_t nb, size_t s);

void bug_hardcodedmemsize()
{
    size_t i, s;

    s = 4;
    int **matrix = (int **)calloc(SIZE20, s);
    if (matrix == NULL) {
        return; /* Indicate calloc() failure */
    }
    fill_ints(matrix, SIZE20, s);
    free(matrix);
}
```

In this example, the memory allocation function `calloc` is called with a memory size of 4. The memory is allocated for an integer pointer, which can be a more or less than 4 bytes depending on your target. If the integer pointer is not 4 bytes, your program can fail.

Correction — Use `sizeof(int *)`

When calling `calloc`, replace the hard-coded size with a call to `sizeof`. This change makes your code more portable.

```
#include <stddef.h>
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};
extern void fill_ints(int **matrix, size_t nb, size_t s);

void corrected_hardcodedmemsize()
{
    size_t i, s;

    s = sizeof(int *);
    int **matrix = (int **)calloc(SIZE20, s);
    if (matrix == NULL) {
        return; /* Indicate calloc() failure */
    }
    fill_ints(matrix, SIZE20, s);
    free(matrix);
}
```

Check Information

Group: Rec. 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP09-C

Introduced in R2019a

CERT C: Rec. EXP10-C

Do not depend on the order of evaluation of subexpressions or the order in which side effects take place

Description

Rule Definition

Do not depend on the order of evaluation of subexpressions or the order in which side effects take place.

Polyspace Implementation

This checker checks for **Expression value depends on order of evaluation or of side effects**.

Examples

Expression value depends on order of evaluation or of side effects

Issue

The issue occurs when the value of an expression and its persistent side effects is not the same under all permitted evaluation orders.

An expression can have different values under the following conditions:

- The same variable is modified more than once in the expression, or is both read and written.
- The expression allows more than one order of evaluation.

Therefore, this rule forbids expressions where a variable is modified more than once and can cause different results under different orders of evaluation.

Risk

If an expression results in different values depending on the order of evaluation, its value becomes implementation-defined.

Example - Variable Modified More Than Once in Expression

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);         /* Compliant */
    COPY_ELEMENT (i++);       /* Noncompliant  */
}
```

In this example, the rule is violated by the statement `COPY_ELEMENT(i++)` because `i++` occurs twice and the order of evaluation of the two expressions is unspecified.

Example - Variable Modified and Used in Multiple Function Arguments

```
void f (unsigned int param1, unsigned int param2) {}

void main () {
    unsigned int i=0;
    f ( i++, i );                /* Non-compliant */
}
```

In this example, the rule is violated because it is unspecified whether the operation `i++` occurs before or after the second argument is passed to `f`. The call `f(i++,i)` can translate to either `f(0,0)` or `f(0,1)`.

Check Information

Group: Rec. 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

"Check for Coding Standard Violations"

External Websites

EXP10-C

Introduced in R2019a

CERT C: Rec. EXP12-C

Do not ignore values returned by functions

Description

Rule Definition

Do not ignore values returned by functions.

Polyspace Implementation

This checker checks for **Returned value of a sensitive function not checked**.

Examples

Returned value of a sensitive function not checked

Issue

Returned value of a sensitive function not checked occurs when you call sensitive standard functions, but you:

- Ignore the return value.
- Use an output or a return value without testing the validity of the return value.

For this defect, two type of functions are considered: ***sensitive*** and ***critical sensitive***.

A ***sensitive*** function is a standard function that can encounter:

- Exhausted system resources (for example, when allocating resources)
- Changed privileges or permissions
- Tainted sources when reading, writing, or converting data from external sources
- Unsupported features despite an existing API

A ***critical sensitive*** function is a sensitive function that performs one of these critical or vulnerable tasks:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

Risk

If you do not check the return value of functions that perform sensitive or critical sensitive tasks, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

Fix

Before continuing with the program, test the return value of *critical sensitive* functions.

For *sensitive functions*, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

Example - Sensitive Function Return Ignored

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    pthread_attr_init(&attr);
}
```

This example shows a call to the sensitive function `pthread_attr_init`. The return value of `pthread_attr_init` is ignored, causing a defect.

Correction — Cast Function to `(void)`

One possible correction is to cast the function to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of the sensitive function.

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    (void)pthread_attr_init(&attr);
}
```

Correction — Test Return Value

One possible correction is to test the return value of `pthread_attr_init` to check for errors.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

void initialize() {
    pthread_attr_t attr;
    int result;

    result = pthread_attr_init(&attr);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

Example - Critical Function Return Ignored

```
#include <pthread.h>
extern void *start_routine(void *);

void returnnotchecked() {
```

```
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0));
    pthread_join(thread_id,  &res);
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), Polyspace does not ignore this *Return value of a sensitive function not checked* defect. The other critical function, `pthread_join`, returns value that is ignored implicitly. `pthread_join` uses the return value of `pthread_create`, which was not checked.

Correction — Test the Return Value of Critical Functions

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

Check Information

Group: Rec. 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites
EXP12-C

Introduced in R2019a

CERT C: Rec. EXP13-C

Treat relational and equality operators as if they were nonassociative

Description

Rule Definition

Treat relational and equality operators as if they were nonassociative.

Polyspace Implementation

This checker checks for **Possibly unintended evaluation of expression because of operator precedence rules**.

Examples

Possibly unintended evaluation of expression because of operator precedence rules

Issue

Possibly unintended evaluation of expression because of operator precedence rules occurs when an arithmetic expression result is possibly unintended because operator precedence rules dictate an evaluation order that you do not expect.

The defect highlights expressions of the form `x op_1 y op_2 z`. Here, *op_1* and *op_2* are operator combinations that commonly induce this error. For instance, `x == y | z`.

The checker does not flag all operator combinations. For instance, `x == y || z` is not flagged because you most likely intended to perform a logical OR between `x == y` and `z`. Specifically, the checker flags these combinations:

- && and ||: For instance, `x || y && z` or `x && y || z`.
- Assignment and bitwise operations: For instance, `x = y | z`.
- Assignment and comparison operations: For instance, `x = y != z` or `x = y > z`.
- Comparison operations: For instance, `x > y > z` (except when one of the comparisons is an equality `x == y > z`).
- Shift and numerical operation: For instance, `x << y + 2`.
- Pointer dereference and arithmetic: For instance, `*p++`.

Risk

The defect can cause the following issues:

- If you or another code reviewer reviews the code, the intended order of evaluation is not immediately clear.
- It is possible that the result of the evaluation does not meet your expectations. For instance:
  - In the operation `*p++`, it is possible that you expect the dereferenced value to be incremented. However, the pointer `p` is incremented before the dereference.

- In the operation `(x == y | z)`, it is possible that you expect `x` to be compared with `y | z`. However, the == operation happens before the | operation.

Fix

See if the order of evaluation is what you intend. If not, apply parentheses to implement the evaluation order that you want.

For better readability of your code, it is good practice to apply parenthesis to implement an evaluation order even when operator precedence rules impose that order.

Example - Expressions with Possibly Unintended Evaluation Order

```
int test(int a, int b, int c) {
    return(a & b == c);
}
```

In this example, the == operation happens first, followed by the & operation. If you intended the reverse order of operations, the result is not what you expect.

Correction — Parenthesis For Intended Order

One possible correction is to apply parenthesis to implement the intended evaluation order.

```
int test(int a, int b, int c) {
    return((a & b) == c);
}
```

Check Information

Group: Rec. 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP13-C

Introduced in R2019a

CERT C: Rec. EXP15-C

Do not place a semicolon on the same line as an if, for, or while statement

Description

Rule Definition

Do not place a semicolon on the same line as an if, for, or while statement.

Polyspace Implementation

This checker checks for **Semicolon on same line as if, for or while statement**.

Examples

Semicolon on same line as if, for or while statement

Issue

Semicolon on same line as if, for or while statement occurs when a semicolon on the same line as the last token of an `if`, `for` or `while` statement results in an empty body.

The checker makes an exception for the case where the `if` statement is immediately followed by an `else` statement:

```
if(condition);
else {
  ...
}
```

Risk

The semicolon following the `if`, `for` or `while` statement often indicates a programming error. The spurious semicolon changes the execution flow and leads to unintended results.

Fix

If you want an empty body for the `if`, `for` or `while` statement , wrap the semicolon in a block and place the block on a new line to explicitly indicate your intent:

```
if(condition)
   {;}
```

Otherwise, remove the spurious semicolon.

Example - Spurious Semicolon

```
int credentialsOK(void);

void login () {
    int loggedIn = 0;
    if(credentialsOK());
      loggedIn = 1;
}
```

In this example, the spurious semicolon results in an empty `if` body. The assignment `loggedIn=1` is always performed. However, the assignment was probably to be performed only under a condition.

Correction – Remove Spurious Semicolon

If the semicolon was unintended, remove the semicolon.

```
int credentialsOK(void);

void login () {
    int loggedIn = 0;
    if(credentialsOK())
      loggedIn = 1;
}
```

Check Information

Group: Rec. 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP15-C

Introduced in R2020a

CERT C: Rec. EXP19-C

Use braces for the body of an if, for, or while statement

Description

Rule Definition

Use braces for the body of an if, for, or while statement.

Polyspace Implementation

This checker checks for **Iteration or selection statement body not enclosed in braces**.

Examples

Iteration or selection statement body not enclosed in braces

Issue

The issue occurs when you do not enclose the body of an iteration-statement or a selection-statement in braces.

Risk

The rule applies to:

- Iteration statements such as `while`, `do ... while` or `for`.
- Selection statements such as `if ... else` or `switch`.

If the block of code associated with an iteration or selection statement is not contained in braces, you can make mistakes about the association. For example:

- You can wrongly associate a line of code with an iteration or selection statement because of its indentation.
- You can accidentally place a semicolon following the iteration or selection statement. Because of the semicolon, the line following the statement is no longer associated with the statement even though you intended otherwise.

Example - Iteration Block

```
int data_available = 1;
void f1(void) {
    while(data_available)                /* Non-compliant */
        process_data();

    while(data_available) {              /* Compliant */
        process_data();
    }
}
```

In this example, the second `while` block is enclosed in braces and does not violate the rule.

Example - Nested Selection Statements

```
void f1(void) {
    if(flag_1)                          /* Non-compliant */
        if(flag_2)                      /* Non-compliant */
            action_1();
    else                                /* Non-compliant */
            action_2();
}
```

In this example, the rule is violated because the `if` or `else` blocks are not enclosed in braces. Unless indented as above, it is easy to associate the `else` statement with the inner `if`.

Correction — Place Selection Statement Block in Braces

One possible correction is to enclose each block associated with an `if` or `else` statement in braces.

```
void f1(void) {
    if(flag_1) {                        /* Compliant */
        if(flag_2) {                      /* Compliant */
            action_1();
        }
    }
    else {                              /* Compliant */
        action_2();
    }
}
```

Example - Spurious Semicolon After Iteration Statement

```
void f1(void) {
    while(flag_1);                      /* Non-compliant */
    {
        flag_1 = action_1();
    }
}
```

In this example, the rule is violated even though the `while` statement is followed by a block in braces. The semicolon following the `while` statement causes the block to dissociated from the `while` statement.

The rule helps detect such spurious semicolons.

Check Information

Group: Rec. 03. Expressions (EXP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

EXP19-C

Introduced in R2019a

CERT C: Rec. INT00-C

Understand the data model used by your implementation(s)

Description

Rule Definition

Understand the data model used by your implementation(s).

Polyspace Implementation

This checker checks for these issues:

- **Use of basic types declarations and definitions of variables or functions**.
- **Integer overflow**.
- **Integer constant overflow**.
- **Format string specifiers and arguments mismatch**.

Examples

Use of basic types declarations and definitions of variables or functions

Issue

The issue occurs when you use basic numerical types instead of `typedefs` that indicate size and signedness.

The rule checker flags use of basic data types in variable or function declarations and definitions. The rule enforces use of `typedef`s instead.

The rule checker does not flag the use of basic types in the `typedef` statements themselves.

Risk

When the amount of memory being allocated is important, using specific-length types makes it clear how much storage is being reserved for each object.

Example - Direct Use of Basic Types in Definitions

```
typedef unsigned int uint32_t;

int x = 0;       /* Non compliant */
uint32_t y = 0;  /* Compliant */
```

In this example, the declaration of `x` is noncompliant because it uses a basic type directly.

Integer overflow

Issue

Integer overflow occurs when an operation on integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Integer overflows on signed integers result in undefined behavior.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

To avoid overflows in general, try one of these techniques:

- Keep integer variable values restricted to within half the range of signed integers.
- In operations that might overflow, check for conditions that can lead to the overflow and implement wrap around or saturation behavior depending on how the result of the operation is used. The result then becomes predictable and can be safely used in subsequent computations.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Addition of Maximum Integer

```
#include <limits.h>

int plusplus(void) {

    int var = INT_MAX;
    var++;
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

Correction — Different Storage Type

One possible correction is to change data types. Store the result of the operation in a larger data type (Note that on a 32-bit machine, `int` and `long` has the same size). In this example, on a 32-bit machine, by returning a `long long` instead of an `int`, the overflow error is fixed.

```
#include <limits.h>

long long plusplus(void) {

    long long lvar = INT_MAX;
```

```
    lvar++;
    return lvar;
}
```

Integer constant overflow

Issue

Integer constant overflow occurs when you assign a compile-time constant to a signed integer variable whose data type cannot accommodate the value. An n-bit signed integer holds values in the range $[-2^{n-1}, 2^{n-1}-1]$.

For instance, `c` is an 8-bit signed `char` variable that cannot hold the value 255.

```
signed char c = 255;
```

To determine the sizes of fundamental types, Bug Finder uses your specification for `Target processor type (-target)`.

Risk

The default behavior for constant overflows can vary between compilers and platforms. Retaining constant overflows can reduce the portability of your code.

Even if your compilers wraps around overflowing constants with a warning, the wrap-around behavior can be unintended and cause unexpected results.

Fix

Check if the constant value is what you intended. If the value is correct, use a different, possibly wider, data type for the variable.

Example - Overflowing Constant from Macro Expansion

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
    char c1 = MAX_UNSIGNED_CHAR;
    char c2 = MAX_SIGNED_CHAR+1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow. To reproduce these defects, use a `Target processor type (-target)` where `char` is signed by default.

Correction — Use Different Data Type

One possible correction is to use a different data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
    unsigned char c1 = MAX_UNSIGNED_CHAR;
    unsigned char c2 = MAX_SIGNED_CHAR+1;
}
```

Format string specifiers and arguments mismatch

Issue

Format string specifiers and arguments mismatch occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
  const char *message = "License not available";
  int err_number = ;-4
  printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Printing a Float

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", fst);
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

Check Information

Group: Rec. 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT00-C

Introduced in R2019a

CERT C: Rec. INT02-C

Understand integer conversion rules

Description

Rule Definition

Understand integer conversion rules.

Polyspace Implementation

This checker checks for **Sign change integer conversion overflow**.

Examples

Sign change integer conversion overflow

Issue

Sign change integer conversion overflow occurs when converting an unsigned integer to a signed integer. If the variable does not have enough bytes to represent both the original constant and the sign bit, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Convert from `unsigned char` to `char`

```
char sign_change(void) {
    unsigned char count = 255;

    return (char)count;
}
```

In the return statement, the unsigned character variable `count` is converted to a signed character. However, `char` has 8 bits, 1 for the sign of the constant and 7 to represent the number. The conversion operation overflows because 255 uses 8 bits.

Correction — Change conversion types

One possible correction is using a larger integer type. By using an `int`, there are enough bits to represent the sign and the number value.

```
int sign_change(void) {
    unsigned char count = 255;

    return (int)count;
}
```

Check Information

Group: Rec. 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT02-C

Introduced in R2019a

CERT C: Rec. INT04-C

Enforce limits on integer values originating from tainted sources

Description

Rule Definition

Enforce limits on integer values originating from tainted sources.

Polyspace Implementation

This checker checks for these issues:

- **Array access with tainted index**.
- **Loop bounded with tainted value**.
- **Memory allocation with tainted size**.
- **Tainted size of variable length array**.

Examples

Array access with tainted index

Issue

Array access with tainted index detects reading or writing to an array by using a tainted index that has not been validated.

Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

Example - Use Index to Return Buffer Value

```
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    return tab[num];
}
```

In this example, the index `num` accesses the array `tab`. The function does not check to see if `num` is inside the range of `tab`.

Correction — Check Range Before Use

One possible correction is to check that `num` is in range before using it.

```
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    if (num >= 0 && num < SIZE100) {
           return tab[num];
    } else {
        return -9999;
    }
}
```

Loop bounded with tainted value

Issue

Loop bounded with tainted value detects loops that are bounded by values from an unsecure source.

Risk

A tainted value can cause over looping or infinite loops. Attackers can use this vulnerability to crash your program or cause other unintended behavior.

Fix

Before starting the loop, validate unknown boundary and iterator values.

Example - Loop Boundary From Input Argument

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedloopboundary(int count) {
    int res = 0;
    for (int i=0 ; i < count; ++i) {
        res += i;
    }
    return res;
}
```

In this example, the function uses the input argument to loop `count` times. `count` could be any number because the value is not checked before starting the for-loop.

Correction — Check Loop Control

One possible correction is to check the value of the variable controlling the loop before starting the for-loop. This example checks if `count` is greater than zero and less than the maximum size.

```
enum {
    SIZE10  =  10,
```

```
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedloopboundary(int count) {
    int res = 0;

    if (count>0 && count<SIZE128) {
        for (int i=0 ; i<count ; ++i) {
            res += i;
        }
    }
    return res;
}
```

Memory allocation with tainted size

Issue

Memory allocation with tainted size checks memory allocation functions, such as `calloc` or `malloc`, for size arguments from unsecured sources.

Risk

Uncontrolled memory allocation can cause your program to request too much system memory. This consequence can lead to a crash due to an out-of-memory condition, or assigning too many resources.

Fix

Before allocating memory, check the value of your arguments to check that they do not exceed the bounds.

Example - Allocate Memory Using Input Argument

```
#include "stdlib.h"

int* bug_taintedmemoryallocsize(size_t size) {
    int* p = (int*)malloc(size);
    return p;
}
```

In this example, `malloc` allocates `size` amount of memory for the pointer `p`. `size` is an outside variable, so could be any size value. If the size is larger than the amount of memory you have available, your program could crash.

Correction — Check Size of Memory to be Allocated

One possible correction is to check the size of the memory that you want to allocate before performing the `malloc` operation. This example checks to see if the size is positive and less than the maximum size.

```
#include "stdlib.h"

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
```

```
};

int* corrected_taintedmemoryallocsize(int size) {
    int* p = NULL;
    if (size>0 && size<SIZE128) {          /* Fix: Check entry range before use */
        p = (int*)malloc((unsigned int)size);
    }
    return p;
}
```

Tainted size of variable length array

Issue

Tainted size of variable length array detects variable length arrays (VLA) whose size is from an unsecure source.

Risk

If an attacker changed the size of your VLA to an unexpected value, it can cause your program to crash or behave unexpectedly.

If the size is non-positive, the behavior of the VLA is undefined. Your program does not perform as expected.

If the size is unbounded, the VLA can cause memory exhaustion or stack overflow.

Fix

Validate your VLA size to make sure that it is positive and less than a maximum value.

Example - Input Argument Used as Size of VLA

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedvlasize(int size) {

    int tabvla[size];
    int res = 0;
    for (int i=0 ; i<SIZE10 ; ++i) {
        tabvla[i] = i*i;
        res += tabvla[i];
    }
    return res;
}
```

In this example, a variable length array size is based on an input argument. Because this input argument value is not checked, the size may be negative or too large.

Correction — Check VLA Size

One possible correction is to check the size variable before creating the variable length array. This example checks if the size is larger than 10 and less than 100, before creating the VLA

```
enum {
    SIZE10  =  10,
```

```
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedvlasize(int size) {
    int res = 0;
    if (size>SIZE10 && size<SIZE100) {
        int tabvla[size];
        for (int i=0 ; i<SIZE10 ; ++i) {
            tabvla[i] = i*i;
            res += tabvla[i];
        }
    }
    return res;
}
```

Check Information

Group: Rec. 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT04-C

Introduced in R2019a

CERT C: Rec. INT07-C

Use only explicitly signed or unsigned char type for numeric values

Description

Rule Definition

Use only explicitly signed or unsigned char type for numeric values.

Polyspace Implementation

This checker checks for **Use of plain char type for numerical value**.

Examples

Use of plain char type for numerical value

Issue

Use of plain char type for numerical value detects `char` variables without explicit signedness that are being used in these ways:

- To store non-char constants
- In an arithmetic operation when the `char` is:
  - A negative value.
  - The result of a sign changing overflow.
- As a buffer offset.

`char` variables without a `signed` or `unsigned` qualifier can be either signed or unsigned depending on your compiler.

Risk

Operations on a plain char can result in unexpected numerical values. If the char is used as an offset, the char can cause buffer overflow or underflow.

Fix

When initializing a char variable, to avoid implementation-defined confusion, explicitly state whether the char is signed or unsigned.

Example - Divide by `char` Variable

```
#include <stdio.h>

void badplaincharuse(void)
{
    char c = 200;
    int i = 1000;
    (void)printf("i/c = %d\n", i/c);
}
```

In this example, the char variable `c` can be signed or unsigned depending on your compiler. Assuming 8-bit, two's complement character types, the result is either `i/c = 5` (unsigned char) or `i/c = -17` (signed char). The correct result is unknown without knowing the signedness of `char`.

Correction — Add `signed` Qualifier

One possible correction is to add a `signed` qualifier to `char`. This clarification makes the operation defined.

```
#include <stdio.h>

void badplaincharuse(void)
{
    signed char c = -56;
    int i = 1000;
    (void)printf("i/c = %d\n", i/c);
}
```

Check Information

Group: Rec. 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT07-C

Introduced in R2019a

CERT C: Rec. INT08-C

Verify that all integer values are in range

Description

Rule Definition

Verify that all integer values are in range.

Polyspace Implementation

This checker checks for these issues:

- **Integer overflow**.
- **Integer constant overflow**.

Examples

Integer overflow

Issue

Integer overflow occurs when an operation on integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Integer overflows on signed integers result in undefined behavior.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

To avoid overflows in general, try one of these techniques:

- Keep integer variable values restricted to within half the range of signed integers.
- In operations that might overflow, check for conditions that can lead to the overflow and implement wrap around or saturation behavior depending on how the result of the operation is used. The result then becomes predictable and can be safely used in subsequent computations.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Addition of Maximum Integer

```
#include <limits.h>

int plusplus(void) {

    int var = INT_MAX;
    var++;
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

Correction — Different Storage Type

One possible correction is to change data types. Store the result of the operation in a larger data type (Note that on a 32-bit machine, `int` and `long` has the same size). In this example, on a 32-bit machine, by returning a `long long` instead of an `int`, the overflow error is fixed.

```
#include <limits.h>

long long plusplus(void) {

    long long lvar = INT_MAX;
    lvar++;
    return lvar;
}
```

Integer constant overflow

Issue

Integer constant overflow occurs when you assign a compile-time constant to a signed integer variable whose data type cannot accommodate the value. An `n`-bit signed integer holds values in the range $[-2^{n-1}, 2^{n-1}-1]$.

For instance, `c` is an 8-bit signed `char` variable that cannot hold the value 255.

```
signed char c = 255;
```

To determine the sizes of fundamental types, Bug Finder uses your specification for `Target processor type (-target)`.

Risk

The default behavior for constant overflows can vary between compilers and platforms. Retaining constant overflows can reduce the portability of your code.

Even if your compilers wraps around overflowing constants with a warning, the wrap-around behavior can be unintended and cause unexpected results.

Fix

Check if the constant value is what you intended. If the value is correct, use a different, possibly wider, data type for the variable.

Example - Overflowing Constant from Macro Expansion

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
    char c1 = MAX_UNSIGNED_CHAR;
    char c2 = MAX_SIGNED_CHAR+1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow. To reproduce these defects, use a `Target processor type (-target)` where `char` is signed by default.

Correction — Use Different Data Type

One possible correction is to use a different data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
    unsigned char c1 = MAX_UNSIGNED_CHAR;
    unsigned char c2 = MAX_SIGNED_CHAR+1;
}
```

Check Information

Group: Rec. 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT08-C

Introduced in R2019a

CERT C: Rec. INT09-C

Ensure enumeration constants map to unique values

Description

Rule Definition

Ensure enumeration constants map to unique values.

Polyspace Implementation

This checker checks for **Enumeration constants map to same value**.

Examples

Enumeration constants map to same value

Issue

The issue occurs when, within an enumerator list, the value of an implicitly-specified enumeration constants are not unique.

The rule checker flags an enumeration if it has an implicitly specified enumeration constant with the same value as another enumeration constant.

Risk

An implicitly specified enumeration constant has a value one greater than its predecessor. If the first enumeration constant is implicitly specified, then its value is 0. An explicitly specified enumeration constant has the specified value.

If implicitly and explicitly specified constants are mixed within an enumeration list, it is possible for your program to replicate values. Such replications can be unintentional and can cause unexpected behavior.

Example - Replication of Value in Implicitly Specified Enum Constants

```
enum color1 {red_1, blue_1, green_1};   /* Compliant */
enum color2 {red_2 = 1, blue_2 = 2, green_2 = 3};       /* Compliant */
enum color3 {red_3 = 1, blue_3, green_3};     /* Compliant */
enum color4 {red_4, blue_4, green_4 = 1};     /* Non Compliant */
enum color5 {red_5 = 2, blue_5, green_5 = 2};     /* Compliant */
enum color6 {red_6 = 2, blue_6, green_6 = 2, yellow_6};     /* Non Compliant */
```

Compliant situations:

- `color1`: All constants are implicitly specified.
- `color2`: All constants are explicitly specified.
- `color3`: Though there is a mix of implicit and explicit specification, all constants have unique values.
- `color5`: The implicitly specified constants have unique values.

Noncompliant situations:

- `color4`: The implicitly specified constant `blue_4` has the same value as `green_4`.
- `color6`: The implicitly specified constant `blue_6` has the same value as `yellow_6`.

Check Information

Group: Rec. 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT09-C

Introduced in R2019a

CERT C: Rec. INT10-C

Do not assume a positive remainder when using the % operator

Description

Rule Definition

Do not assume a positive remainder when using the % operator.

Polyspace Implementation

This checker checks for **Tainted modulo operand**.

Examples

Tainted modulo operand

Issue

Tainted modulo operand checks the operands of remainder `%` operations. Bug Finder flags modulo operations with one or more tainted operands.

Risk

- If the second remainder operand is zero, your remainder operation fails, causing your program to crash.
- If the second remainder operand is `-1`, your remainder operation can overflow if the remainder operation is implemented based on the division operation that can overflow.
- If one of the operands is negative, the operation result is uncertain. For C89, the modulo operation is not standardized, so the result from negative operands is implementation-defined.

These risks can be exploited by attackers to gain access to your program or the target in general.

Fix

Before performing the modulo operation, validate the values of the operands. Check the second operand for values of `0` and `-1`. Check both operands for negative values.

Example - Modulo with Function Arguments

```
extern void print_int(int);

int taintedintmod(int userden) {
    int rem =  128%userden;
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using an input argument. The argument is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

Correction — Check Operand Values

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
extern void print_int(int);

int taintedintmod(int userden) {
    int rem = 0;
    if (userden > 0) {
        rem = 128 % userden;
    }
    print_int(rem);
    return rem;
}
```

Check Information

Group: Rec. 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT10-C

Introduced in R2019a

CERT C: Rec. INT12-C

Do not make assumptions about the type of a plain int bit-field when used in an expression

Description

Rule Definition

Do not make assumptions about the type of a plain int bit-field when used in an expression.

Polyspace Implementation

This checker checks for **Bit-field declared without appropriate type**.

Examples

Bit-field declared without appropriate type

Issue

The issue occurs when you declare a bit-filed without an appropriate type.

Risk

Using `int` is implementation-defined because bit-fields of type `int` can be either `signed` or `unsigned`.

The use of `enum`, `short char`, or any other type of bit-field is not permitted in C90 because the behavior is undefined.

In C99, the implementation can potentially define other integer types that are permitted in bit-field declarations.

Check Information

Group: Rec. 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT12-C

Introduced in R2019a

CERT C: Rec. INT13-C

Use bitwise operators only on unsigned operands

Description

Rule Definition

Use bitwise operators only on unsigned operands.

Polyspace Implementation

This checker checks for **Bitwise operation on negative value**.

Examples

Bitwise operation on negative value

Issue

Bitwise operation on negative value detects bitwise operators (>>, ^, |, ~, `but`, `not`, &) used on signed integer variables with negative values.

Risk

If the value of the signed integer is negative, bitwise operation results can be unexpected because:

- Bitwise operations on negative values are compiler-specific.
- Unexpected calculations can lead to additional vulnerabilities, such as buffer overflow.

Fix

When performing bitwise operations, use `unsigned` integers to avoid unexpected results.

Example - Right-Shift of Negative Integer

```
#include <stdio.h>
#include <stdarg.h>

static void demo_sprintf(const char *format, ...)
{
    int rc;
    va_list ap;
    char buf[sizeof("256")];

    va_start(ap, format);
    rc = vsprintf(buf, format, ap);
    if (rc == -1 || rc >= sizeof(buf)) {
        /* Handle error */
    }
    va_end(ap);
}

void bug_bitwiseneg()
{
```

```
    int stringify = 0x80000000;
    demo_sprintf("%u", stringify >> 24);
}
```

In this example, the statement `demo_sprintf("%u", stringify >> 24)` stops the program unexpectedly. You expect the result of `stringify >> 24` to be `0x80`. However, the actual result is `0xffffff80` because `stringify` is signed and negative. The sign bit is also shifted.

Correction — Add `unsigned` Keyword

By adding the `unsigned` keyword, `stringify` is not negative and the right-shift operation gives the expected result of `0x80`.

```
#include <stdio.h>
#include <stdarg.h>

static void demo_sprintf(const char *format, ...)
{
    int rc;
    va_list ap;
    char buf[sizeof("256")];

    va_start(ap, format);
    rc = vsprintf(buf, format, ap);
    if (rc == -1 || rc >= sizeof(buf)) {
        /* Handle error */
    }
    va_end(ap);
}

void corrected_bitwiseneg()
{
    unsigned int stringify = 0x80000000;
    demo_sprintf("%u", stringify >> 24);
}
```

Check Information

Group: Rec. 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT13-C

Introduced in R2019a

CERT C: Rec. INT14-C

Avoid performing bitwise and arithmetic operations on the same data

Description

Rule Definition

Avoid performing bitwise and arithmetic operations on the same data.

Polyspace Implementation

This checker checks for **Bitwise and arithmetic operation on the same data**.

Examples

Bitwise and arithmetic operation on the same data

Issue

Bitwise and arithmetic operation on a same data detects statements with bitwise and arithmetic operations on the same variable or expression.

Risk

Mixed bitwise and arithmetic operations *do* compile. However, the size of integer types affects the result of these mixed operations. Mixed operations also reduce readability and maintainability.

Fix

Separate bitwise and arithmetic operations, or use only one type of operation per statement.

Example - Shift and Addition

```
unsigned int bitwisearithmix()
{
    unsigned int var = 50;
    var += (var << 2) + 1;
    return var;
}
```

This example shows bitwise and arithmetic operations on the variable `var`. `var` is shifted by two (bitwise), then increased by 1 and added to itself (arithmetic).

Correction — Arithmetic Operations Only

You can reduce this expression to arithmetic-only operations: `var + (var << 2)` is equivalent to `var * 5`.

```
unsigned int bitwisearithmix()
{
    unsigned int var = 50;
    var = var * 5 +1;
    return var;
}
```

Check Information

Group: Rec. 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

INT14-C

Introduced in R2019a

CERT C: Rec. INT18-C

Evaluate integer expressions in a larger size before comparing or assigning to that size

Description

Rule Definition

Evaluate integer expressions in a larger size before comparing or assigning to that size.

Polyspace Implementation

This checker checks for these issues:

- **Integer overflow**.
- **Unsigned integer overflow**.

Examples

Integer overflow

Issue

Integer overflow occurs when an operation on integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Integer overflows on signed integers result in undefined behavior.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

To avoid overflows in general, try one of these techniques:

- Keep integer variable values restricted to within half the range of signed integers.
- In operations that might overflow, check for conditions that can lead to the overflow and implement wrap around or saturation behavior depending on how the result of the operation is used. The result then becomes predictable and can be safely used in subsequent computations.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Addition of Maximum Integer

```
#include <limits.h>

int plusplus(void) {

    int var = INT_MAX;
    var++;
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

Correction — Different Storage Type

One possible correction is to change data types. Store the result of the operation in a larger data type (Note that on a 32-bit machine, `int` and `long` has the same size). In this example, on a 32-bit machine, by returning a `long long` instead of an `int`, the overflow error is fixed.

```
#include <limits.h>

long long plusplus(void) {

    long long lvar = INT_MAX;
    lvar++;
    return lvar;
}
```

Unsigned integer overflow

Issue

Unsigned integer overflow occurs when an operation on unsigned integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

The C11 standard states that unsigned integer overflows result in wrap-around behavior. However, a wrap around behavior might not always be desirable. For instance, if the result of a computation is used as an array size and the computation overflows, the array size is much smaller than expected.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling. In the error handling code, you can override the default wrap-around behavior for overflows and implement saturation behavior, for instance.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Add One to Maximum Unsigned Integer

```
#include <limits.h>

unsigned int plusplus(void) {

    unsigned uvar = UINT_MAX;
    uvar++;
    return uvar;
}
```

In the third statement of this function, the variable `uvar` is increased by 1. However, the value of `uvar` is the maximum unsigned integer value, so 1 plus the maximum integer value cannot be represented by an `unsigned int`. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `uvar` is reduced by modulo `UINT_MAX`. The result is `uvar = 1`.

Correction — Different Storage Type

One possible correction is to store the operation result in a larger data type. In this example, by returning an `unsigned long long` instead of an `unsigned int`, the overflow error is fixed.

```
#include <limits.h>

unsigned long long plusplus(void) {

    unsigned long long ullvar = UINT_MAX;
    ullvar++;
    return ullvar;
}
```

Check Information

Group: Rec. 04. Integers (INT)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

"Check for Coding Standard Violations"

External Websites

INT18-C

Introduced in R2019a

CERT C: Rec. FLP00-C

Understand the limitations of floating-point numbers

Description

Rule Definition

Understand the limitations of floating-point numbers.

Polyspace Implementation

This checker checks for **Absorption of float operand**.

Examples

Absorption of float operand

Issue

Absorption of float operand occurs when one operand of an addition or subtraction operation is *always* negligibly small compared to the other operand. Therefore, the result of the operation is always equal to the value of the larger operand, making the operation redundant.

Risk

Redundant operations waste execution cycles of your processor.

The absorption of a float operand can indicate design issues elsewhere in the code. It is possible that the developer expected a different range for one of the operands and did not expect the redundancy of the operation. However, the operand range is different from what the developer expects because of issues elsewhere in the code.

Fix

See if the operand ranges are what you expect. To see the ranges, place your cursor on the operation.

- If the ranges are what you expect, justify why you have the redundant operation in place. For instance, the code is only partially written and you anticipate other values for one or both of the operands from future unwritten code.

  If you cannot justify the redundant operation, remove it.
- If the ranges are not what you expect, in your code, trace back to see where the ranges come from. To begin your traceback, search for instances of the operand in your code. Browse through previous instances of the operand and determine where the unexpected range originates.

To determine when one operand is negligible compared to the other operand, the defect uses rules based on IEEE 754 standards. To fix the defect, instead of using the actual rules, you can use this heuristic: the ratio of the larger to the smaller operand must be less than 2^{p-1} at least for some values. Here, `p` is equal to 24 for 32-bit precision and 53 for 64-bit precision. To determine the precision, the defect uses your specification for `Target processor type (-target)`.

This defect appears only if one operand is *always* negligibly smaller than the other operand. To see instances of subnormal operands or results, use the check **Subnormal Float** in Polyspace Code Prover.

Example - One Addition Operand Negligibly Smaller Than The Other Operand

```
#include <stdlib.h>

float get_signal(void);
void do_operation(float);

float input_signal1(void) {
    float temp = get_signal();
    if(temp > 0. && temp < 1e-30)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

float input_signal2(void) {
    float temp = get_signal();
    if(temp > 1.)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

void main() {
    float signal1 = input_signal1();
    float signal2 = input_signal2();
    float super_signal = signal1 + signal2;
    do_operation(super_signal);
}
```

In this example, the defect appears on the addition because the operand `signal1` is in the range `(0,1e-30)` but `signal2` is greater than `1`.

Correction — Remove Redundant Operation

One possible correction is to remove the redundant addition operation. In the following corrected code, the operand `signal2` and its associated code is also removed from consideration.

```
#include <stdlib.h>

float get_signal(void);
void do_operation(float);

float input_signal1(void) {
    float temp = get_signal();
    if(temp > 0. && temp < 1e-30)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
```

```
    }
}

void main() {
    float signal1 = input_signal1();
    do_operation(signal1);
}
```

Correction — Verify Operand Range

Another possible correction is to see if the operand ranges are what you expect. For instance, if one of the operand range is not supposed to be negligibly small, fix the issue causing the small range. In the following corrected code, the range `(0,1e-2)` is imposed on `signal2` so that it is not *always* negligibly small as compared to `signal1`.

```
#include <stdlib.h>

float get_signal(void);
void do_operation(float);

float input_signal1(void) {
    float temp = get_signal();
    if(temp > 0. && temp < 1e-2)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

float input_signal2(void) {
    float temp = get_signal();
    if(temp > 1.)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

void main() {
    float signal1 = input_signal1();
    float signal2 = input_signal2();
    float super_signal = signal1 + signal2;
    do_operation(super_signal);
}
```

Check Information

Group: Rec. 05. Floating Point (FLP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites
FLP00-C

Introduced in R2019a

CERT C: Rec. FLP02-C

Avoid using floating-point numbers when precise computation is needed

Description

Rule Definition

Avoid using floating-point numbers when precise computation is needed.

Polyspace Implementation

This checker checks for **Floating point comparison with equality operators**.

Examples

Floating point comparison with equality operators

Issue

Floating point comparison with equality operators occurs when you use an equality (==) or inequality (!=) operation with floating-point numbers.

Polyspace does not raise a defect for an equality or inequality operation with floating-point numbers when:

- The comparison is between two float constants.

  ```
  float flt = 1.0;
  if (flt == 1.1)
  ```

- The comparison is between a constant and a variable that can take a finite, reasonably small number of values.

  ```
  float x;

  int rand = random();
  switch(rand) {
  case 1: x = 0.0; break;
  case 2: x = 1.3; break;
  case 3: x = 1.7; break;
  case 4: x = 2.0; break;
  default: x = 3.5; break; }
  …
  if (x==1.3)
  ```

- The comparison is between floating-point expressions that contain only integer values.

  ```
  float x = 0.0;
  for (x=0.0;x!=100.0;x+=1.0) {
  …
  if (random) break;
  }

  if (3*x+4==2*x-1)
  ```

```
…
if (3*x+4 == 1.3)
```

- One of the operands is `0.0`, unless you use the option flag `-detect-bad-float-op-on-zero`.

  ```
  /* Defect detected when
  you use the option flag */

  if (x==0.0f)
  ```

  If you are running an analysis through the user interface, you can enter this option in the **Other** field, under the **Advanced Settings** node on the **Configuration** pane. See `Other`.

  At the command line, add the flag to your analysis command.

  ```
  polyspace-bug-finder -sources filename ^
  -checkers BAD_FLOAT_OP -detect-bad-float-op-on-zero
  ```

Risk

Checking for equality or inequality of two floating-point values might return unexpected results because floating-point representations are inexact and involve rounding errors.

Fix

Instead of checking for equality of floating-point values:

```
if (val1 == val2)
```

check if their difference is less than a predefined tolerance value (for instance, the value `FLT_EPSILON` defined in `float.h`):

```
#include <float.h>
if(fabs(val1-val2) < FLT_EPSILON)
```

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Floats Inequality in `for`-loop

```
#include <stdio.h>
#include <math.h>
#include <float.h>

void func(void)
{
    float f;
    for (f = 1.0; f != 2.0; f = f + 0.1)
        (void)printf("Value: %f\n", f);
}
```

In this function, the `for`-loop tests the inequality of `f` and the number 2.0 as a stopping mechanism. The number of iterations is difficult to determine, or might be infinite, because of the imprecision in floating-point representation.

Correction — Change the Operator

One possible correction is to use a different operator that is not as strict. For example, an inequality like >= or <=.

```
#include <stdio.h>
#include <math.h>
#include <float.h>

void func(void)
{
    float f;
    for (f = 1.0; f <= 2.0; f = f + 0.1)
        (void)printf("Value: %f\n", f);
}
```

Check Information

Group: Rec. 05. Floating Point (FLP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FLP02-C

Introduced in R2019a

CERT C: Rec. FLP03-C

Detect and handle floating-point errors

Description

Rule Definition

Detect and handle floating-point errors.

Polyspace Implementation

This checker checks for these issues:

- **Float conversion overflow**.
- **Float overflow**.
- **Float division by zero**.

Examples

Float conversion overflow

Issue

Float conversion overflow occurs when converting a floating point number to a smaller floating point data type. If the variable does not have enough memory to represent the original number, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Overflows can result in unpredictable values from computations. The result can be infinity or the maximum finite value depending on the rounding mode used in the implementation. If you use the result of an overflowing conversion in subsequent computations and do not account for the overflow, you can see unexpected results.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variable being converted acquires its current value You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller floating point types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Converting from `double` to `float`

```
float convert(void) {

    double diam = 1e100;
    return (float)diam;
}
```

In the return statement, the variable `diam` of type double (64 bits) is converted to a variable of type float (32 bits). However, the value 1^100 requires more than 32 bits to be precisely represented.

Float overflow

Issue

Float overflow occurs when an operation on floating point variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Overflows can result in unpredictable values from computations. The result can be infinity or the maximum finite value depending on the rounding mode used in the implementation. If you use the result of an overflowing computation in subsequent computations and do not account for the overflow, you can see unexpected results.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, for instance, when you handle infinities in your code, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Multiplication of Floats

```
#include <float.h>

float square(void) {

   float val = FLT_MAX;
   return val * val;
}
```

In the return statement, the variable `val` is multiplied by itself. The square of the maximum float value cannot be represented by a float (the return type for this function) because the value of `val` is the maximum float value.

Correction — Different Storage Type

One possible correction is to store the result of the operation in a larger data type. In this example, by returning a `double` instead of a `float`, the overflow defect is fixed.

```
#include <float.h>

double square(void) {
    float val = FLT_MAX;

    return (double)val * (double)val;
}
```

Float division by zero

Issue

Float division by zero occurs when the denominator of a division operation can be a zero-valued floating point number.

Risk

A division by zero can result in a program crash.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the denominator variable acquires a zero value. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

It is a good practice to check for zero values of a denominator before division and handle the error. Instead of performing the division directly:

```
res = num/den;
```

use a library function that handles zero values of the denominator before performing the division:

```
res = div(num, den);
```

See examples of fixes below.

If you do not want to fix the issue, for instance, when you handle infinities in your code, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Dividing a Floating Point Number by Zero

```
float fraction(float num)
{
    float denom = 0.0;
    float result = 0.0;

    result = num/denom;
```

```
    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

Correction — Check Before Division

```
float fraction(float num)
{
    float denom = 0.0;
    float result = 0.0;

    if( ((int)denom) != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
float fraction(float num)
{
    float denom = 2.0;
    float result = 0.0;

    result = num/denom;

    return result;
}
```

Check Information

Group: Rec. 05. Floating Point (FLP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FLP03-C

Introduced in R2019a

CERT C: Rec. FLP06-C

Convert integers to floating point for floating-point operations

Description

Rule Definition

Convert integers to floating point for floating-point operations.

Polyspace Implementation

This checker checks for **Float overflow**.

Examples

Float overflow

Issue

Float overflow occurs when an operation on floating point variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Overflows can result in unpredictable values from computations. The result can be infinity or the maximum finite value depending on the rounding mode used in the implementation. If you use the result of an overflowing computation in subsequent computations and do not account for the overflow, you can see unexpected results.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, for instance, when you handle infinities in your code, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Multiplication of Floats

```
#include <float.h>

float square(void) {
```

```
   float val = FLT_MAX;
   return val * val;
}
```

In the return statement, the variable `val` is multiplied by itself. The square of the maximum float value cannot be represented by a float (the return type for this function) because the value of `val` is the maximum float value.

Correction — Different Storage Type

One possible correction is to store the result of the operation in a larger data type. In this example, by returning a `double` instead of a `float`, the overflow defect is fixed.

```
#include <float.h>

double square(void) {
    float val = FLT_MAX;

    return (double)val * (double)val;
}
```

Check Information

Group: Rec. 05. Floating Point (FLP)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FLP06-C

Introduced in R2019a

CERT C: Rec. ARR01-C

Do not apply the sizeof operator to a pointer when taking the size of an array

Description

Rule Definition

Do not apply the sizeof operator to a pointer when taking the size of an array.

Polyspace Implementation

This checker checks for these issues:

- **Wrong type used in sizeof**.
- **Possible misuse of sizeof**.

Examples

Wrong type used in sizeof

Issue

Wrong type used in sizeof occurs when both of the following conditions hold:

- You assign the address of a block of memory to a pointer, or transfer data between two blocks of memory. The assignment or copy uses the `sizeof` operator.

  For instance, you initialize a pointer using `malloc(sizeof(`*`type`*`))` or copy data between two addresses using `memcpy(`*`destination_ptr`*`, `*`source_ptr`*`, sizeof(`*`type`*`))`.
- You use an incorrect type as argument of the `sizeof` operator. You use the pointer type instead of the type that the pointer points to.

  For instance, to initialize a *`type`*`*` pointer, you use `malloc(sizeof(`*`type`*`*`))` instead of `malloc(sizeof(`*`type`*`))`.

Risk

Irrespective of what *`type`* stands for, the expression `sizeof(`*`type`*`*`)` always returns a fixed size. The size returned is the pointer size on your platform in bytes. The appearance of `sizeof(`*`type`*`*`)` often indicates an unintended usage. The error can cause allocation of a memory block that is much smaller than what you need and lead to weaknesses such as buffer overflows.

For instance, assume that `structType` is a structure with ten `int` variables. If you initialize a `structType*` pointer using `malloc(sizeof(structType*))` on a 32-bit platform, the pointer is assigned a memory block of four bytes. However, to be allocated completely for one `structType` variable, the `structType*` pointer must point to a memory block of `sizeof(structType) = 10 * sizeof(int)` bytes. The required size is much greater than the actual allocated size of four bytes.

Fix

To initialize a *`type`*`*` pointer, replace `sizeof(`*`type`*`*`)` in your pointer initialization expression with `sizeof(`*`type`*`)`.

Example - Allocate a Char Array With `sizeof`

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char*) * 5);
    free(str);

}
```

In this example, memory is allocated for the character pointer `str` using a `malloc` of five char pointers. However, `str` is a pointer to a character, not a pointer to a character pointer. Therefore the `sizeof` argument, `char*`, is incorrect.

Correction — Match Pointer Type to `sizeof` Argument

One possible correction is to match the argument to the pointer type. In this example, `str` is a character pointer, therefore the argument must also be a character.

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char) * 5);
    free(str);

}
```

Possible misuse of sizeof

Issue

Possible misuse of sizeof occurs when Polyspace Bug Finder detects possibly unintended results from the use of `sizeof` operator. For instance:

- You use the `sizeof` operator on an array parameter name, expecting the array size. However, the array parameter name by itself is a pointer. The `sizeof` operator returns the size of that pointer.
- You use the `sizeof` operator on an array element, expecting the array size. However, the operator returns the size of the array element.
- The size argument of certain functions such as `strncmp` or `wcsncpy` is incorrect because you used the `sizeof` operator earlier with possibly incorrect expectations. For instance:
  - In a function call `strncmp(string1, string2, num)`, `num` is obtained from an incorrect use of the `sizeof` operator on a pointer.
  - In a function call `wcsncpy(destination, source, num)`, `num` is the not the number of wide characters but a size in bytes obtained by using the `sizeof` operator. For instance, you use `wcsncpy(destination, source, sizeof(destination) - 1)` instead of `wcsncpy(destination, source, (sizeof(desintation)/sizeof(wchar_t)) - 1)`.

Risk

Incorrect use of the `sizeof` operator can cause the following issues:

- If you expect the `sizeof` operator to return array size and use the return value to constrain a loop, the number of loop runs are smaller than what you expect.
- If you use the return value of `sizeof` operator to allocate a buffer, the buffer size is smaller than what you require. Insufficient buffer can lead to resultant weaknesses such as buffer overflows.
- If you use the return value of `sizeof` operator incorrectly in a function call, the function does not behave as you expect.

Fix

Possible fixes are:

- Do not use the `sizeof` operator on an array parameter name or array element to determine array size.

  The best practice is to pass the array size as a separate function parameter and use that parameter in the function body.
- Use the `sizeof` operator carefully to determine the number argument of functions such as `strncmp` or `wcsncpy`. For instance, for wide string functions such as `wcsncpy`, use the number of wide characters as argument instead of the number of bytes.

Example - `sizeof` Used Incorrectly to Determine Array Size

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < sizeof(a)/sizeof(int); i++)    {
        a[i] = i + 1;
    }
}
```

In this example, `sizeof(a)` returns the size of the pointer `a` and not the array size.

Correction — Determine Array Size in Another Way

One possible correction is to use another means to determine the array size.

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < MAX_SIZE; i++)    {
        a[i] = i + 1;
    }
}
```

Check Information

Group: Rec. 06. Arrays (ARR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics
“Check for Coding Standard Violations”

External Websites
ARR01-C

Introduced in R2019a

CERT C: Rec. ARR02-C

Explicitly specify array bounds, even if implicitly defined by an initializer

Description

Rule Definition

Explicitly specify array bounds, even if implicitly defined by an initializer.

Polyspace Implementation

This checker checks for the issue **Improper array initialization**.

Examples

Improper array initialization

Issue

Improper array initialization occurs when Polyspace Bug Finder considers that an array initialization using initializers is incorrect.

This defect applies to normal and designated initializers. In C99, with designated initializers, you can place the elements of an array initializer in any order and implicitly initialize some array elements. The designated initializers use the array index to establish correspondence between an array element and an array initializer element. For instance, the statement `int arr[6] = { [4] = 29, [2] = 15 }` is equivalent to `int arr[6] = { 0, 0, 15, 0, 29, 0 }`.

You can use initializers incorrectly in one of the following ways.

Issue	Risk	Possible Fix
In your initializer for a one-dimensional array, you have more elements than the array size.	Unused array initializer elements indicate a possible coding error.	Increase the array size or remove excess elements.
You place the braces enclosing initializer values incorrectly.	Because of the incorrect placement of braces, some array initializer elements are not used. Unused array initializer elements indicate a possible coding error.	Place braces correctly.
In your designated initializer, you do not initialize the first element of the array explicitly.	The implicit initialization of the first array element indicates a possible coding error. You possibly overlooked the fact that array indexing starts from 0.	Initialize all elements explicitly.

Issue	Risk	Possible Fix
In your designated initializer, you initialize an element twice.	The first initialization is overridden. The redundant first initialization indicates a possible coding error.	Remove the redundant initialization.
You use designated and nondesignated initializers in the same initialization.	You or another reviewer of your code cannot determine the size of the array by inspection.	Use either designated or nondesignated initializers.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Incorrectly Placed Braces (C Only)

```
int arr[2][3]
= {{1, 2},
    {3, 4},
    {5, 6}
};
```

In this example, the array `arr` is initialized as `{1,2,0,3,4,0}`. Because the initializer contains `{5,6}`, you might expect the array to be initialized `{1,2,3,4,5,6}`.

Correction — Place Braces Correctly

One possible correction is to place the braces correctly so that all elements are explicitly initialized.

```
int a1[2][3]
= {{1, 2, 3},
    {4, 5, 6}
};
```

Example - First Element Not Explicitly Initialized

```
int arr[5]
= {
    [1] = 2,
    [2] = 3,
    [3] = 4,
    [4] = 5
};
```

In this example, `arr[0]` is not explicitly initialized. It is possible that the programmer did not consider that the array indexing starts from 0.

Correction — Explicitly Initialize All Elements

One possible correction is to initialize all elements explicitly.

```
int arr[5]
= {
    [0] = 1,
    [1] = 2,
    [2] = 3,
    [3] = 4,
    [4] = 5
};
```

Example - Element Initialized Twice

```
int arr[5]
= {
    [0] = 1,
    [1] = 2,
    [2] = 3,
    [2] = 4,
    [4] = 5
};
```

In this example, `arr[2]` is initialized twice. The first initialization is overridden. In this case, because `arr[3]` was not explicitly initialized, it is possible that the programmer intended to initialize `arr[3]` when `arr[2]` was initialized a second time.

Correction — Fix Redundant Initialization

One possible correction is to eliminate the redundant initialization.

```
int arr[5]
= {
    [0] = 1,
    [1] = 2,
    [2] = 3,
    [3] = 4,
    [4] = 5
};
```

Example - Mix of Designated and Nondesignated Initializers

```
int arr[]
= {
    [0] = 1,
    [3] = 3,
    4,
    [5] = 5,
    6
    };
```

In this example, because a mix of designated and nondesignated initializers are used, it is difficult to determine the size of `arr` by inspection.

Correction — Use Only Designated Initializers

One possible correction is to use only designated initializers for array initialization.

```
int arr[]
= {
    [0] = 1,
    [3] = 3,
    [4] = 4,
    [5] = 5,
    [6] = 6
};
```

Check Information

Group: Rec. 06. Arrays (ARR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

ARR02-C

Introduced in R2019a

CERT C: Rec. STR02-C

Sanitize data passed to complex subsystems

Description

Rule Definition

Sanitize data passed to complex subsystems.

Polyspace Implementation

This checker checks for these issues:

- **Execution of externally controlled command**.
- **Command executed from externally controlled path**.
- **Library loaded from externally controlled path**.

Examples

Execution of externally controlled command

Issue

Execution of externally controlled command checks for commands that are fully or partially constructed from externally controlled input.

Risk

Attackers can use the externally controlled input as operating system commands, or arguments to the application. An attacker could read or modify sensitive data can be read or modified, execute unintended code, or gain access to other aspects of the program.

Fix

Validate the inputs to allow only intended input values. For example, create a whitelist of acceptable inputs and compare the input against this list.

Example - Call Argument Command

```
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "unistd.h"
#include "dlfcn.h"
#include "limits.h"

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
```

```
};

void taintedexternalcmd(char* usercmd)
{
    char cmd[SIZE128] = "/usr/bin/cat ";
    strcat(cmd, usercmd);
    system(cmd);
}
```

This example function calls a command from a user argument without checking the command variable.

Correction — Use a Predefined Command

One possible correction is to use a `switch` statement to run a predefined command, using the user input as the switch variable.

```
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "unistd.h"
#include "dlfcn.h"
#include "limits.h"

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
enum { CMD0 = 1, CMD1, CMD2 };

void taintedexternalcmd(int usercmd)
{
    char cmd[SIZE128] = "/usr/bin/cat ";

    switch(usercmd) {
        case CMD0:
            strcat(cmd, "*.c");
            break;
        case CMD1:
            strcat(cmd, "*.h");
            break;
        case CMD2:
            strcat(cmd, "*.cpp");
            break;
        default:
            strcat(cmd, "*.c");
    }
    system(cmd);
}
```

Command executed from externally controlled path

Issue

Command executed from externally controlled path checks the path of commands that the application controls. If the path of a command is from or constructed from external sources, Bug Finder flags the command function.

Risk

An attacker can:

- Change the command that the program executes, possibly to a command that only the attack can control.
- Change the environment in which the command executes, by which the attacker controls what the command means and does.

Fix

Before calling the command, validate the path to make sure that it is the intended location.

Example - Executing Path from Environment Variable

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedpathcmd() {
    char cmd[SIZE128] = "";
    char* userpath = getenv("MYAPP_PATH");

    strncpy(cmd, userpath, SIZE100);
    strcat(cmd, "/ls *");
    /* Launching command */
    system(cmd);
}
```

This example obtains a path from an environment variable `MYAPP_PATH`. `system` runs a command from that path without checking the value of the path. If the path is not the intended path, your program executes in the wrong location.

Correction — Use Trusted Path

One possible correction is to use a list of allowed paths to match against the environment variable path.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

enum {
    SIZE10  =  10,
```

```
    SIZE100 = 100,
    SIZE128 = 128
};

/* Function to sanitize a string */
int sanitize_str(char* s, size_t n) {
    int res = 0;
    /* String is ok if */
    if (s && n>0 && n<SIZE128) {
        /* - string is not null                       */
        /* - string has a positive and limited size */
        s[n-1] = '\0';  /* Add a security \0 char at end of string */
        /* Tainted pointer detected above, used as "firewall" */
        res = 1;
    }
    return res;
}

/* Authorized path ids */
enum { PATH0=1, PATH1, PATH2 };

void taintedpathcmd() {
    char cmd[SIZE128] = "";

    char* userpathid = getenv("MYAPP_PATH_ID");
    if (sanitize_str(userpathid, SIZE100)) {
        int pathid = atoi(userpathid);


        char path[SIZE128] = "";
        switch(pathid) {
            case PATH0:
                strcpy(path, "/usr/local/my_app0");
                break;
            case PATH1:
                strcpy(path, "/usr/local/my_app1");
                break;
            case PATH2:
                strcpy(path, "/usr/local/my_app2");
                break;
            default:
                /* do nothing */
        break;
        }
        if (strlen(path)>0) {
            strncpy(cmd, path, SIZE100);
            strcat(cmd, "/ls *");
            system(cmd);
        }
    }
}
```

Library loaded from externally controlled path

Issue

Library loaded from externally controlled path looks for libraries loaded from fixed or controlled paths. If unintended actors can control one or more locations on this fixed path, Bug Finder raises a defect.

Risk

If an attacker knows or controls the path that you use to load a library, the attacker can change:

- The library that the program loads, replacing the intended library and commands.
- The environment in which the library executes, giving unintended permissions and capabilities to the attacker.

Fix

When possible, use hard-coded or fully qualified path names to load libraries. It is possible the hard-coded paths do not work on other systems. Use a centralized location for hard-coded paths, so that you can easily modify the path within the source code.

Another solution is to use functions that require explicit paths. For example, `system()` does not require a full path because it can use the `PATH` environment variable. However, `execl()` and `execv()` do require the full path.

Example - Call Custom Library

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dlfcn.h>
#include <limits.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void* taintedpathlib() {
    void* libhandle = NULL;
    char lib[SIZE128] = "";
    char* userpath = getenv("LD_LIBRARY_PATH");
    strncpy(lib, userpath, SIZE128);
    strcat(lib, "/libX.so");
    libhandle = dlopen(lib, 0x00001);
    return libhandle;
}
```

This example loads the library `libX.so` from an environment variable `LD_LIBRARY_PATH`. An attacker can change the library path in this environment variable. The actual library you load could be a different library from the one that you intend.

Correction — Change and Check Path

One possible correction is to change how you get the library path and check the path of the library before opening the library. This example receives the path as an input argument. Then the path is checked to make sure the library is not under `/usr/`.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dlfcn.h>
#include <limits.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

/* Function to sanitize a string */
int sanitize_str(char* s, size_t n) {
    /* strlen is used here as a kind of firewall for tainted string errors */
    int res = (strlen(s) > 0 && strlen(s) < n);
    return res;
}
void* taintedpathlib(char* userpath) {
    void* libhandle = NULL;
    if (sanitize_str(userpath, SIZE128)) {
        char lib[SIZE128] = "";

        if (strncmp(userpath, "/usr", 4)!=0) {
            strncpy(lib, userpath, SIZE128);
            strcat(lib, "/libX.so");
            libhandle = dlopen(lib, RTLD_LAZY);
        }
    }
    return libhandle;
}
```

Check Information

Group: Rec. 07. Characters and Strings (STR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

STR02-C

Introduced in R2019a

CERT C: Rec. STR03-C

Do not inadvertently truncate a string

Description

Rule Definition

Do not inadvertently truncate a string.

Polyspace Implementation

This checker checks for **Invalid use of standard library string routine**.

Examples

Invalid use of standard library string routine

Issue

Invalid use of standard library string routine occurs when a string library function is called with invalid arguments.

Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Invalid Use of Standard Library String Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_String(void)
{
 char *res;
 char gbuffer[5],text[20]="ABCDEFGHIJKL";
```

```
  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

Check Information

Group: Rec. 07. Characters and Strings (STR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

STR03-C

Introduced in R2019a

CERT C: Rec. STR07-C

Use the bounds-checking interfaces for string manipulation

Description

Rule Definition

Use the bounds-checking interfaces for string manipulation.

Polyspace Implementation

This checker checks for these issues:

- **Use of dangerous standard function**.
- **Destination buffer overflow in string manipulation**.

Examples

Use of dangerous standard function

Issue

The **Use of dangerous standard function** check highlights uses of functions that are inherently dangerous or potentially dangerous given certain circumstances. The following table lists possibly dangerous functions, the risks of using each function, and what function to use instead.

Dangerous Function	Risk Level	Safer Function
`gets`	Inherently dangerous — You cannot control the length of input from the console.	`fgets`
`cin`	Inherently dangerous — You cannot control the length of input from the console.	Avoid or prefaces calls to `cin` with `cin.width`.
`strcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`strncpy`
`stpcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`stpncpy`
`lstrcpy` or `StrCpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy`
`strcat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`strncat`, `strlcat`, or `strcat_s`

Dangerous Function	Risk Level	Safer Function
`lstrcat` or `StrCat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`StringCbCat`, `StringCchCat`, `strncay`, `strcat_s`, or `strlcat`
`wcpcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`wcpncpy`
`wcscat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`wcsncat`, `wcslcat`, or `wcncat_s`
`wcscpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`wcsncpy`
`sprintf`	Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur.	`snprintf`
`vsprintf`	Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur.	`vsnprintf`

Risk

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Using `sprintf`

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (sprintf(dst, "%s", str) == 1)
    {
```

```
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

Correction — Use `snprintf` with Buffer Size

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

Destination buffer overflow in string manipulation

Issue

Destination buffer overflow in string manipulation occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.

- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

Example - Buffer Overflow in `sprintf` Use

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

Check Information

Group: Rec. 07. Characters and Strings (STR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

STR07-C

Introduced in R2019a

CERT C: Rec. STR11-C

Do not specify the bound of a character array initialized with a string literal

Description

Rule Definition

Do not specify the bound of a character array initialized with a string literal.

Polyspace Implementation

This checker checks for **Missing null in string array**.

Examples

Missing null in string array

Issue

Missing null in string array occurs when a string does not have enough space to terminate with a null character `'\0'`.

This defect applies only for projects in C.

Risk

A buffer overflow can occur if you copy a string to an array without assuming the implicit null terminator.

Fix

If you initialize a character array with a literal, avoid specifying the array bounds.

```
char three[]  = "THREE";
```

The compiler automatically allocates space for a null terminator. In the preceding example, the compiler allocates sufficient space for five characters and a null terminator.

If the issue occurs after initialization, you might have to increase the size of the array by one to account for the null terminator.

In certain circumstances, you might want to initialize the character array with a sequence of characters instead of a string. In this situation, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Array size is too small

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[5] = "THREE";
}
```

The character array `three` has a size of 5 and 5 characters `'T'`, `'H'`, `'R'`, `'E'`, and `'E'`. There is no room for the null character at the end because `three` is only five bytes large.

Correction — Increase Array Size

One possible correction is to change the array size to allow for the five characters plus a null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[6] = "THREE";
}
```

Correction — Change Initialization Method

One possible correction is to initialize the string by leaving the array size blank. This initialization method allocates enough memory for the five characters and a terminating-null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[]  = "THREE";
}
```

Check Information

Group: Rec. 07. Characters and Strings (STR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

STR11-C

Introduced in R2019a

CERT C: Rec. MEM00-C

Allocate and free memory in the same module, at the same level of abstraction

Description

Rule Definition

Allocate and free memory in the same module, at the same level of abstraction.

Polyspace Implementation

This checker checks for these issues:

- **Invalid free of pointer**.
- **Deallocation of previously deallocated pointer**.
- **Use of previously freed pointer**.

Examples

Invalid free of pointer

Issue

Invalid free of pointer occurs when a block of memory released using the `free` function was not previously allocated using `malloc`, `calloc`, or `realloc`.

Risk

The `free` function releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.

The issue can highlight coding errors. For instance, you perhaps wanted to use the `free` function or a previous `malloc` function on a different pointer.

Fix

In most cases, you can fix the issue by removing the `free` statement. If the pointer is not allocated memory from the heap with `malloc` or `calloc`, you do not need to free the pointer. You can simply reuse the pointer as required.

If the issue highlights a coding error such as use of `free` or `malloc` on the wrong pointer, correct the error.

If the issue occurs because you use the `free` function to free memory allocated with the `new` operator, replace the `free` function with the `delete` operator.

Example - Invalid Free of Pointer Error

```
#include <stdlib.h>

void Assign_Ones(void)
{
```

```
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;

  free(p);
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer `p` is deallocated using the `free` function. However, `p` points to a memory location that was not dynamically allocated.

Correction — Remove Pointer Deallocation

If the number of elements of the array `p` is known at compile time, one possible correction is to remove the deallocation of the pointer `p`.

```
#include <stdlib.h>

void Assign_Ones(void)
 {
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;
  /* Fix: Remove deallocation of p */
 }
```

Correction — Introduce Pointer Allocation

If the number of elements of the array `p` is not known at compile time, one possible correction is to dynamically allocate memory to the array `p`.

```
#include <stdlib.h>

void Assign_Ones(int num)
{
  int *p;
  /* Fix: Allocate memory dynamically to p */
  p=(int*) calloc(10,sizeof(int));
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
}
```

Deallocation of previously deallocated pointer

Issue

Deallocation of previously deallocated pointer occurs when a block of memory is freed more than once using the `free` function without an intermediate allocation.

Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to free this block of memory can result in a segmentation fault.

Fix

The fix depends on the root cause of the defect. See if you intended to allocate a memory block to the pointer between the first deallocation and the second. Otherwise, remove the second `free` statement.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before freeing pointers, check them for NULL values and handle the error. In this way, you are protected against freeing an already freed block.

Example - Deallocation of Previously Deallocated Pointer Error

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    free (pi);
    /* Defect: pi has already been freed */
}
```

The first `free` statement releases the block of memory that `pi` refers to. The second `free` statement on `pi` releases a block of memory that has been freed already.

Correction — Remove Duplicate Deallocation

One possible correction is to remove the second `free` statement.

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    /* Fix: remove second deallocation */
 }
```

Use of previously freed pointer

Issue

Use of previously freed pointer occurs when you access a block of memory after freeing the block using the `free` function.

Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to access this block of memory can result in unpredictable behavior or even a segmentation fault.

Fix

The fix depends on the root cause of the defect. See if you intended to free the memory later or allocate another memory block to the pointer before access.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before dereferencing pointers, check them for NULL values and handle the error. In this way, you are protected against accessing a freed block.

Example - Use of Previously Freed Pointer Error

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   {
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);

    j = *pi + shift;
    /* Defect: Reading a freed pointer */

    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, dereferencing`pi` after the `free` statement is not valid.

Correction — Free Pointer After Use

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
#include <stdlib.h>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

Check Information

Group: Rec. 08. Memory Management (MEM)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MEM00-C

Introduced in R2019a

CERT C: Rec. MEM01-C

Store a new value in pointers immediately after free()

Description

Rule Definition

Store a new value in pointers immediately after free().

Polyspace Implementation

This checker checks for **Missing reset of a freed pointer**.

Examples

Missing reset of a freed pointer

Issue

Missing reset of a freed pointer detects pointers that have been freed and not reassigned another value. After freeing a pointer, the memory data is still accessible. To clear this data, the pointer must also be set to NULL or another value.

Risk

Not resetting pointers can cause dangling pointers. Dangling pointers can cause:

- Freeing already freed memory.
- Reading from or writing to already freed memory.
- Hackers executing code stored in freed pointers or with vulnerable permissions.

Fix

After freeing a pointer, if it is not immediately assigned to another valid address, set the pointer to NULL.

Example - Free Without Reset

```
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};

void missingfreedptrreset()
{
    static char *str = NULL;

    if (str == NULL)
        str = (char *)malloc(SIZE20);

    if (str != NULL)
```

```
        free(str);
}
```

In this example, the pointer `str` is freed at the end of the program. The next call to `bug_missingfreedptrrese` can fail because `str` is not NULL and the initialization to NULL can be invalid.

Correction — Redefine `free` to Free and Reset

One possible correction is to customize `free` so that when you free a pointer, it is automatically reset.

```
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};

static void sanitize_free(void **p)
{
    if ((p != NULL) && (*p != NULL))
    {
        free(*p);
        *p = NULL;
    }
}

#define free(X) sanitize_free((void **)&X)

void missingfreedptrreset()
{
    static char *str = NULL;

    if (str == NULL)
        str = (char *)malloc(SIZE20);

    if (str != ((void *)0))
    {
        free(str);
    }
}
```

Check Information

Group: Rec. 08. Memory Management (MEM)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MEM01-C

Introduced in R2019a

CERT C: Rec. MEM02-C

Immediately cast the result of a memory allocation function call into a pointer to the allocated type

Description

Rule Definition

Immediately cast the result of a memory allocation function call into a pointer to the allocated type.

Polyspace Implementation

This checker checks for **Wrong allocated object size for cast**.

Examples

Wrong allocated object size for cast

Issue

Wrong allocated object size for cast occurs during pointer conversion when the pointer's address is misaligned. If a pointer is converted to a different pointer type, the size of the allocated memory must be a multiple of the size of the destination pointer.

Risk

Dereferencing a misaligned pointer has undefined behavior and can cause your program to crash.

Fix

Suppose you convert a pointer `ptr1` to `ptr2`. If `ptr1` points to a buffer of `N` bytes and `ptr2` is a *`type`* * pointer where `sizeof(`*`type`*`)` is `n` bytes, make sure that `N` is an integer multiple of `n`.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Dynamic Allocation of Pointers

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(13);
    long *dest;

    dest = (long*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to a `long`*. The dynamically allocated memory of `ptr`, 13 bytes, is not a multiple of the size of `dest`, 4 bytes. This misalignment causes the **Wrong allocated object size for cast** defect.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the allocated memory to 12 instead of 13.

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(12);
    long *dest;

    dest = (long*)ptr;
}
```

Example - Static Allocation of Pointers

```
void static_non_align(void){
    char arr[13], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to an `int*` in line 6. `ptr` has a memory size of 13 bytes because the array `arr` has a size of 13 bytes. The size of `dest` is 4 bytes, which is not a multiple of 13. This misalignment causes the **Wrong allocated object size for cast** defect.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the size of the array `arr` to a multiple of 4.

```
void static_non_align(void){
    char arr[12], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr;
}
```

Example - Allocation with a Function

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(13);  //defect
    dest2 = (char*)my_alloc(13); //not a defect
}
```

In this example, the software raises a defect on the conversion of the pointer returned by `my_alloc(13)` to an `int*` in line 11. `my_alloc(13)` returns a pointer with a dynamically allocated size of 13 bytes. The size of `dest1` is 4 bytes, which is not a divisor of 13. This misalignment causes the **Wrong allocated object size for cast** defect. In line 12, the same function call, `my_alloc(13)`, does not call a defect for the conversion to `dest2` because the size of `char*`, 1 byte, a divisor of 13.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the argument for `my_alloc` to a multiple of 4.

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(12);
    dest2 = (char*)my_alloc(13);
}
```

Check Information

Group: Rec. 08. Memory Management (MEM)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MEM02-C

Introduced in R2019a

CERT C: Rec. MEM03-C

Clear sensitive information stored in reusable resources

Description

Rule Definition

Clear sensitive information stored in reusable resources.

Polyspace Implementation

This checker checks for these issues:

- **Sensitive heap memory not cleared before release**.
- **Uncleared sensitive data in stack**.

Examples

Sensitive heap memory not cleared before release

Issue

Sensitive heap memory not cleared before release detects dynamically allocated memory containing sensitive data. If you do not clear the sensitive data when you free the memory, Bug Finder raises a defect on the `free` function.

Risk

If the memory zone is reallocated, an attacker can still inspect the sensitive data in the old memory zone.

Fix

Before calling `free`, clear out the sensitive data using `memset` or `SecureZeroMemory`.

Example - Sensitive Buffer Freed, Not Cleared

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>

void sensitiveheapnotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char* buf = (char*) malloc(1024);
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    free(buf);
}
```

In this example, the function uses a buffer of passwords and frees the memory before the end of the function. However, the data in the memory is not cleared by using the `free` command.

Correction — Nullify Data

One possible correction is to write over the data to clear out the sensitive information. This example uses `memset` to write over the data with zeros.

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>
#include <assert.h>

#define isNull(arr) for(int i=0;i<(sizeof(arr)/sizeof(arr[0]));i++) assert(arr[i]==0)

void sensitiveheapnotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char* buf = (char*) malloc(1024);

    if (buf) {
        getpwnam_r(my_user, &pwd, buf, bufsize, &result);
        memset(buf, 0, (size_t)1024);
        isNull(buf);
        free(buf);
    }
}
```

Uncleared sensitive data in stack

Issue

Uncleared sensitive data in stack detects static memory containing sensitive data. If you do not clear the sensitive data from your stack before exiting the function or program, Bug Finder raises a defect on the last curly brace.

Risk

Leaving sensitive information in your stack, such as passwords or user information, allows an attacker additional access to the information after your program has ended.

Fix

Before exiting a function or program, clear out the memory zones that contain sensitive data by using `memset` or `SecureZeroMemory`.

Example - Static Buffer of Password Information

```
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>

void bug_sensitivestacknotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
}
```

In this example, a static buffer is filled with password information. The program frees the stack memory at the end of the program. However, the data is still accessible from the memory.

Correction — Clear Memory

One possible correction is to write over the memory before exiting the function. This example uses `memset` to clear the data from the buffer memory.

```
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>
#include <assert.h>

#define isNull(arr) for(int i=0; i<(sizeof(arr)/sizeof(arr[0])); i++) assert(arr[i]==0)

void corrected_sensitivestacknotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    memset(buf, 0, (size_t)1024);
    isNull(buf);
}
```

Check Information

Group: Rec. 08. Memory Management (MEM)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MEM03-C

Introduced in R2019a

CERT C: Rec. MEM04-C

Beware of zero-length allocations

Description

Rule Definition

Beware of zero-length allocations.

Polyspace Implementation

This checker checks for these issues:

- **Variable length array with nonpositive size**.
- **Tainted sign change conversion**.
- **Tainted size of variable length array**.

Examples

Variable length array with nonpositive size

Issue

Variable length array with non-positive size occurs when size of a variable-length array is zero or negative.

Risk

If the size of a variable-length array is zero or negative, unexpected behavior can occur, such as stack overflow.

Fix

When you declare a variable-length array as a local variable in a function:

- If you use a function parameter as the array size, check that the parameter is positive.
- If you use the result of a computation on a function parameter as the array size, check that the result is positive.

You can place a test for positive value either before the function call or the array declaration in the function body.

Example - Nonpositive Array Size

```
int input(void);

void add_scalar(int n, int m) {
    int r=0;
    int arr[m][n];
    for (int i=0; i<m; i++) {
        for (int j=0; j<n; j++) {
            arr[i][j] = input();
            r += arr[i][j];
```

```
        }
    }
}

void main() {
    add_scalar(2,2);
    add_scalar(-1,2);
    add_scalar(2,0);
}
```

In this example, the second and third calls to `add_scalar` result in a negative and zero size of `arr`.

Correction — Make Array Size Positive

One possible correction is fix or remove calls that result in a nonpositive array size.

Tainted sign change conversion

Issue

Tainted sign change conversion looks for values from unsecure sources that are converted, implicitly or explicitly, from signed to unsigned values.

For example, functions that use `size_t` as arguments implicitly convert the argument to an unsigned integer. Some functions that implicitly convert `size_t` are:

```
bcmp
memcpy
memmove
strncmp
strncpy
calloc
malloc
memalign
```

Risk

If you convert a small negative number to unsigned, the result is a large positive number. The large positive number can create security vulnerabilities. For example, if you use the unsigned value in:

- Memory size routines — causes allocating memory issues.
- String manipulation routines — causes buffer overflow.
- Loop boundaries — causes infinite loops.

Fix

To avoid converting unsigned negative values, check that the value being converted is within an acceptable range. For example, if the value represents a size, validate that the value is not negative and less than the maximum value size.

Example - Set Memory Value with Size Argument

```
#include <stdlib.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
```

```
    SIZE128 = 128
};

void bug_taintedsignchange(int size) {
    char str[SIZE128] = "";
    if (size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

In this example, a `char` buffer is created and filled using `memset`. The size argument to `memset` is an input argument to the function.

The call to `memset` implicitly converts `size` to unsigned integer. If `size` is a large negative number, the absolute value could be too large to represent as an integer, causing a buffer overflow.

Correction — Check Value of `size`

One possible correction is to check if `size` is inside the valid range. This correction checks if `size` is greater than zero and less than the buffer size before calling `memset`.

```
#include <stdlib.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void corrected_taintedsignchange(int size) {
    char str[SIZE128] = "";
    if (size>0 && size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

Tainted size of variable length array

Issue

Tainted size of variable length array detects variable length arrays (VLA) whose size is from an unsecure source.

Risk

If an attacker changed the size of your VLA to an unexpected value, it can cause your program to crash or behave unexpectedly.

If the size is non-positive, the behavior of the VLA is undefined. Your program does not perform as expected.

If the size is unbounded, the VLA can cause memory exhaustion or stack overflow.

Fix

Validate your VLA size to make sure that it is positive and less than a maximum value.

Example - Input Argument Used as Size of VLA

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedvlasize(int size) {

    int tabvla[size];
    int res = 0;
    for (int i=0 ; i<SIZE10 ; ++i) {
        tabvla[i] = i*i;
        res += tabvla[i];
    }
    return res;
}
```

In this example, a variable length array size is based on an input argument. Because this input argument value is not checked, the size may be negative or too large.

Correction — Check VLA Size

One possible correction is to check the size variable before creating the variable length array. This example checks if the size is larger than 10 and less than 100, before creating the VLA

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedvlasize(int size) {
    int res = 0;
    if (size>SIZE10 && size<SIZE100) {
        int tabvla[size];
        for (int i=0 ; i<SIZE10 ; ++i) {
            tabvla[i] = i*i;
            res += tabvla[i];
        }
    }
    return res;
}
```

Check Information

Group: Rec. 08. Memory Management (MEM)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MEM04-C

Introduced in R2019a

CERT C: Rec. MEM05-C

Avoid large stack allocations

Description

Rule Definition

Avoid large stack allocations.

Polyspace Implementation

This checker checks for these issues:

- **Direct or indirect function call to itself**.
- **Variable length array with nonpositive size**.
- **Tainted size of variable length array**.

Examples

Direct or indirect function call to itself

Issue

The issue occurs when your code contains functions that call themselves directly or indirectly.

Risk

Variables local to a function are stored in the call stack. If a function calls itself directly or indirectly several times, the available stack space can be exceeded, causing serious failure. Unless the recursion is tightly controlled, it is difficult to determine the maximum stack space required.

Example - Direct and Indirect Recursion

```
void foo1( void ) {     /* Non-compliant - Indirect recursion foo1->foo2->foo1... */
    foo2();
    foo1();               /* Non-compliant - Direct recursion */
}

void foo2( void ) {
    foo1();
}
```

In this example, the rule is violated because of:

- Direct recursion `foo1` → `foo1`.
- Indirect recursion `foo1` → `foo2` → `foo1`.

Variable length array with nonpositive size

Issue

Variable length array with non-positive size occurs when size of a variable-length array is zero or negative.

Risk

If the size of a variable-length array is zero or negative, unexpected behavior can occur, such as stack overflow.

Fix

When you declare a variable-length array as a local variable in a function:

- If you use a function parameter as the array size, check that the parameter is positive.
- If you use the result of a computation on a function parameter as the array size, check that the result is positive.

You can place a test for positive value either before the function call or the array declaration in the function body.

Example - Nonpositive Array Size

```
int input(void);

void add_scalar(int n, int m) {
    int r=0;
    int arr[m][n];
    for (int i=0; i<m; i++) {
        for (int j=0; j<n; j++) {
            arr[i][j] = input();
            r += arr[i][j];
        }
    }
}

void main() {
    add_scalar(2,2);
    add_scalar(-1,2);
    add_scalar(2,0);
}
```

In this example, the second and third calls to `add_scalar` result in a negative and zero size of `arr`.

Correction — Make Array Size Positive

One possible correction is fix or remove calls that result in a nonpositive array size.

Tainted size of variable length array

Issue

Tainted size of variable length array detects variable length arrays (VLA) whose size is from an unsecure source.

Risk

If an attacker changed the size of your VLA to an unexpected value, it can cause your program to crash or behave unexpectedly.

If the size is non-positive, the behavior of the VLA is undefined. Your program does not perform as expected.

If the size is unbounded, the VLA can cause memory exhaustion or stack overflow.

Fix

Validate your VLA size to make sure that it is positive and less than a maximum value.

Example - Input Argument Used as Size of VLA

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedvlasize(int size) {

    int tabvla[size];
    int res = 0;
    for (int i=0 ; i<SIZE10 ; ++i) {
        tabvla[i] = i*i;
        res += tabvla[i];
    }
    return res;
}
```

In this example, a variable length array size is based on an input argument. Because this input argument value is not checked, the size may be negative or too large.

Correction — Check VLA Size

One possible correction is to check the size variable before creating the variable length array. This example checks if the size is larger than 10 and less than 100, before creating the VLA

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedvlasize(int size) {
    int res = 0;
    if (size>SIZE10 && size<SIZE100) {
        int tabvla[size];
        for (int i=0 ; i<SIZE10 ; ++i) {
            tabvla[i] = i*i;
            res += tabvla[i];
        }
    }
    return res;
}
```

Check Information

Group: Rec. 08. Memory Management (MEM)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites
MEM05-C

Introduced in R2019a

CERT C: Rec. MEM06-C

Ensure that sensitive data is not written out to disk

Description

Rule Definition

Ensure that sensitive data is not written out to disk.

Polyspace Implementation

This checker checks for **Sensitive data printed out**.

Examples

Sensitive data printed out

Issue

Sensitive data printed out detects print functions, such as `stdout` or `stderr`, that print sensitive information.

The checker considers the following as sensitive information:

- Return values of password manipulation functions such as `getpw`, `getpwnam` or `getpwuid`.
- Input values of functions such as the Windows-specific function `LogonUser`.

Risk

Printing sensitive information, such as passwords or user information, allows an attacker additional access to the information.

Fix

One fix for this defect is to not print out sensitive information.

If you are saving your logfile to an external file, set the file permissions so that attackers cannot access the logfile information.

Example - Printing Passwords

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

extern void verify_null(const char* buf);
void bug_sensitivedataprint(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
```

```
    puts("Name\n");
    puts(pwd.pw_name);
    puts("PassWord\n");
    puts(pwd.pw_passwd);
    memset(buf, 0, sizeof(buf));
    verify_null(buf);
}
```

In this example, Bug Finder flags `puts` for printing out the password `pwd.pw_passwd`.

Correction — Obfuscate the Password

One possible correction is to obfuscate the password information so that the information is not visible.

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

extern void verify_null(const char* buf);

void sensitivedataprint(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    puts("Name\n");
    puts(pwd.pw_name);
    puts("PassWord\n");
    puts("XXXXXXXX\n");
    memset(buf, 0, sizeof(buf));
    verify_null(buf);
}
```

Check Information

Group: Rec. 08. Memory Management (MEM)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MEM06-C

Introduced in R2019a

CERT C: Rec. MEM11-C

Do not assume infinite heap space

Description

Rule Definition

Do not assume infinite heap space.

Polyspace Implementation

This checker checks for **Unprotected dynamic memory allocation**.

Examples

Unprotected dynamic memory allocation

Issue

Unprotected dynamic memory allocation occurs when you do not check after dynamic memory allocation whether the memory allocation succeeded.

Risk

When memory is dynamically allocated using `malloc`, `calloc`, or `realloc`, it returns a value `NULL` if the requested memory is not available. If the code following the allocation accesses the memory block without checking for this `NULL` value, this access is not protected from failures.

Fix

Check the return value of `malloc`, `calloc`, or `realloc` for NULL before accessing the allocated memory location.

```
int *ptr = malloc(size * sizeof(int));

if(ptr) /* Check for NULL */
{
   /* Memory access through ptr */
}
```

Example - Unprotected dynamic memory allocation error

```
#include <stdlib.h>

void Assign_Value(void)
{
  int* p = (int*)calloc(5, sizeof(int));

  *p = 2;
  /* Defect: p is not checked for NULL value */

  free(p);
}
```

If the memory allocation fails, the function `calloc` returns `NULL` to `p`. Before accessing the memory through `p`, the code does not check whether `p` is `NULL`

Correction — Check for NULL Value

One possible correction is to check whether `p` has value `NULL` before dereference.

```
#include <stdlib.h>

void Assign_Value(void)
 {
   int* p = (int*)calloc(5, sizeof(int));

   /* Fix: Check if p is NULL */
   if(p!=NULL) *p = 2;

   free(p);
 }
```

Check Information

Group: Rec. 08. Memory Management (MEM)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MEM11-C

Introduced in R2019a

CERT C: Rec. MEM12-C

Consider using a goto chain when leaving a function on error when using and releasing resources

Description

Rule Definition

Consider using a goto chain when leaving a function on error when using and releasing resources.

Polyspace Implementation

This checker checks for these issues:

- **Memory leak**.
- **Resource leak**.

Examples

Memory leak

Issue

Memory leak occurs when you do not free a block of memory allocated through `malloc`, `calloc`, `realloc`, or `new`. If the memory is allocated in a function, the defect does not occur if:

- Within the function, you free the memory using `free` or `delete`.
- The function returns the pointer assigned by `malloc`, `calloc`, `realloc`, or `new`.
- The function stores the pointer in a global variable or in a parameter.

Risk

Dynamic memory allocation functions such as `malloc` allocate memory on the heap. If you do not release the memory after use, you reduce the amount of memory available for another allocation. On embedded systems with limited memory, you might end up exhausting available heap memory even during program execution.

Fix

Determine the scope where the dynamically allocated memory is accessed. Free the memory block at the end of this scope.

To free a block of memory, use the `free` function on the pointer that was used during memory allocation. For instance:

```
ptr = (int*)malloc(sizeof(int));
...
free(ptr);
```

It is a good practice to allocate and free memory in the same module at the same level of abstraction. For instance, in this example, `func` allocates and frees memory at the same level but `func2` does not.

```
void func() {
  ptr = (int*)malloc(sizeof(int));
```

```
  {
    ...
  }
  free(ptr);
}

void func2() {
  {
   ptr = (int*)malloc(sizeof(int));
   ...
  }
  free(ptr);
}
```

See CERT-C Rule MEM00-C.

Example - Dynamic Memory Not Released Before End of Function

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }


    *pi = 42;
    /* Defect: pi is not freed */
}
```

In this example, `pi` is dynamically allocated by `malloc`. The function `assign_memory` does not free the memory, nor does it return `pi`.

Correction — Free Memory

One possible correction is to free the memory referenced by `pi` using the `free` function. The `free` function must be called before the function `assign_memory` terminates

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }
    *pi = 42;

    /* Fix: Free the pointer pi*/
    free(pi);
}
```

Correction — Return Pointer from Dynamic Allocation

Another possible correction is to return the pointer `pi`. Returning `pi` allows the function calling `assign_memory` to free the memory block using `pi`.

```
#include<stdlib.h>
#include<stdio.h>

int* assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
            printf("Memory allocation failed");
            return(pi);
        }
    *pi = 42;

    /* Fix: Return the pointer pi*/
    return(pi);
}
```

Example - Memory Leak with New/Delete

```
#define NULL '\0'

void initialize_arr1(void)
{
    int *p_scalar = new int(5);
}

void initialize_arr2(void)
{
    int *p_array = new int[5];
}
```

In this example, the functions create two variables, `p_scalar` and `p_array`, using the `new` keyword. However, the functions end without cleaning up the memory for these pointers. Because the functions used `new` to create these variables, you must clean up their memory by calling `delete` at the end of each function.

Correction — Add Delete

To correct this error, add a `delete` statement for every `new` initialization. If you used brackets `[]` to instantiate a variable, you must call delete with brackets as well.

```
#define NULL '\0'

void initialize_arrs(void)
{
    int *p_scalar = new int(5);
    int *p_array = new int[5];

    delete p_scalar;
    p_scalar = NULL;
```

```
    delete[] p_array;
    p_scalar = NULL;
}
```

Resource leak

Issue

Resource leak occurs when you open a file stream by using a `FILE` pointer but do not close it before:

- The end of the pointer's scope.
- Assigning the pointer to another stream.

Risk

If you do not release file handles explicitly as soon as possible, a failure can occur due to exhaustion of resources.

Fix

Close a `FILE` pointer before the end of its scope, or before you assign the pointer to another stream.

Example - `FILE` Pointer Not Released Before End of Scope

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

In this example, the file pointer `fp1` is pointing to a file `data1.txt`. Before `fp1` is explicitly dissociated from the file stream of `data1.txt`, it is used to access another file `data2.txt`.

Correction — Release `FILE` Pointer

One possible correction is to explicitly dissociate `fp1` from the file stream of `data1.txt`.

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );
    fclose(fp1);

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

Check Information

Group: Rec. 08. Memory Management (MEM)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MEM12-C

Introduced in R2019a

CERT C: Rec. FIO02-C

Canonicalize path names originating from tainted sources

Description

Rule Definition

Canonicalize path names originating from tainted sources.

Polyspace Implementation

This checker checks for **Vulnerable path manipulation**.

Examples

Vulnerable path manipulation

Issue

Vulnerable path manipulation detects relative or absolute path traversals. If the path traversal contains a tainted source, or you use the path to open/create files, Bug Finder raises a defect.

Risk

Relative path elements, such as `".."` can resolve to locations outside the intended folder. Absolute path elements, such as `"/abs/path"` can also resolve to locations outside the intended folder.

An attacker can use these types of path traversal elements to traverse to the rest of the file system and access other files or folders.

Fix

Avoid vulnerable path traversal elements such as `/../` and `/abs/path/`. Use fixed file names and locations wherever possible.

Example - Relative Path Traversal

```
# include <stdio.h>
# include <string.h>
# include <wchar.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <unistd.h>
# include <stdlib.h>
# define BASEPATH "/tmp/"
# define FILENAME_MAX 512

static void Relative_Path_Traversal(void)
{
    char * data;
    char data_buf[FILENAME_MAX] = BASEPATH;
    char sub_buf[FILENAME_MAX];
```

```
    if (fgets(sub_buf, FILENAME_MAX, stdin) == NULL) exit (1);
    data = data_buf;
    strcat(data, sub_buf);

    FILE *file = NULL;
    file = fopen(data, "wb+");
    if (file != NULL) fclose(file);
}

int path_call(void){
    Relative_Path_Traversal();
}
```

This example opens a file from `"/tmp/"`, but uses a relative path to the file. An external user can manipulate this relative path when `fopen` opens the file.

Correction — Use Fixed File Name

One possible correction is to use a fixed file name instead of a relative path. This example uses `file.txt`.

```
# include <stdio.h>
# include <string.h>
# include <wchar.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <unistd.h>
# include <stdlib.h>
# define BASEPATH "/tmp/"
# define FILENAME_MAX 512

static void Relative_Path_Traversal(void)
{
    char * data;
    char data_buf[FILENAME_MAX] = BASEPATH;
    data = data_buf;

    /* FIX: Use a fixed file name */
    strcat(data, "file.txt");
    FILE *file = NULL;
    file = fopen(data, "wb+");
    if (file != NULL) fclose(file);
}

int path_call(void){
    Relative_Path_Traversal();
}
```

Check Information

Group: Rec. 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites
FIO02-C

Introduced in R2019a

CERT C: Rec. FIO11-C

Take care when specifying the mode parameter of fopen()

Description

Rule Definition

Take care when specifying the mode parameter of fopen().

Polyspace Implementation

This checker checks for **Bad file access mode or status**.

Examples

Bad file access mode or status

Issue

Bad file access mode or status occurs when you use functions in the `fopen` or `open` group with invalid or incompatible file access modes, file creation flags, or file status flags as arguments. For instance, for the `open` function, examples of valid:

- Access modes include `O_RDONLY`, `O_WRONLY`, and `O_RDWR`
- File creation flags include `O_CREAT`, `O_EXCL`, `O_NOCTTY`, and `O_TRUNC`.
- File status flags include `O_APPEND`, `O_ASYNC`, `O_CLOEXEC`, `O_DIRECT`, `O_DIRECTORY`, `O_LARGEFILE`, `O_NOATIME`, `O_NOFOLLOW`, `O_NONBLOCK`, `O_NDELAY`, `O_SHLOCK`, `O_EXLOCK`, `O_FSYNC`, `O_SYNC` and so on.

The defect can occur in the following situations.

Situation	Risk	Fix
You pass an empty or invalid access mode to the `fopen` function. According to the ANSI C standard, the valid access modes for `fopen` are: • `r`,`r+` • `w`,`w+` • `a`,`a+` • `rb`, `wb`, `ab` • `r+b`, `w+b`, `a+b` • `rb+`, `wb+`, `ab+`	`fopen` has undefined behavior for invalid access modes. Some implementations allow extension of the access mode such as: • GNU: `rb+cmxe,ccs=utf` • Visual C++: `a+t`, where `t` specifies a text mode. However, your access mode string must begin with one of the valid sequences.	Pass a valid access mode to `fopen`.

Situation	Risk	Fix
You pass the status flag `O_APPEND` to the `open` function without combining it with either `O_WRONLY` or `O_RDWR`.	`O_APPEND` indicates that you intend to add new content at the end of a file. However, without `O_WRONLY` or `O_RDWR`, you cannot write to the file. The `open` function does not return -1 for this logical error.	Pass either `O_APPEND|O_WRONLY` or `O_APPEND|O_RDWR` as access mode.
You pass the status flags `O_APPEND` and `O_TRUNC` together to the `open` function.	`O_APPEND` indicates that you intend to add new content at the end of a file. However, `O_TRUNC` indicates that you intend to truncate the file to zero. Therefore, the two modes cannot operate together. The `open` function does not return -1 for this logical error.	Depending on what you intend to do, pass one of the two modes.
You pass the status flag `O_ASYNC` to the `open` function.	On certain implementations, the mode `O_ASYNC` does not enable signal-driven I/O operations.	Use the `fcntl(pathname, F_SETFL, O_ASYNC);` instead.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Invalid Access Mode with `fopen`

```
#include <stdio.h>

void func(void) {
    FILE *file = fopen("data.txt", "rw");
    if(file!=NULL) {
        fputs("new data",file);
        fclose(file);
    }
}
```

In this example, the access mode `rw` is invalid. Because `r` indicates that you open the file for reading and `w` indicates that you create a new file for writing, the two access modes are incompatible.

Correction — Use Either `r` or `w` as Access Mode

One possible correction is to use the access mode corresponding to what you intend to do.

```
#include <stdio.h>
```

```
void func(void) {
    FILE *file = fopen("data.txt", "w");
    if(file!=NULL) {
        fputs("new data",file);
        fclose(file);
    }
}
```

Check Information

Group: Rec. 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FIO11-C

Introduced in R2019a

CERT C: Rec. FIO21-C

Do not create temporary files in shared directories

Description

Rule Definition

Do not create temporary files in shared directories.

Polyspace Implementation

This checker checks for **Use of non-secure temporary file**.

Examples

Use of non-secure temporary file

Issue

Use of non-secure temporary file looks for temporary file routines that are not secure.

Risk

If an attacker guesses the file name generated by a standard temporary file routine, the attacker can:

- Cause a race condition when you generate the file name.
- Precreate a file of the same name, filled with malicious content. If your program reads the file, the attacker's file can inject the malicious code.
- Create a symbolic link to a file storing sensitive data. When your program writes to the temporary file, the sensitive data is deleted.

Fix

To create temporary files, use a more secure standard temporary file routine, such as `mkstemp` from POSIX.1-2001.

Also, when creating temporary files with routines that allow flags, such as `mkostemp`, use the exclusion flag `O_EXCL` to avoid race conditions.

Example - Temp File Created With `tempnam`

```
#define _BSD_SOURCE
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int test_temp()
```

```
{
    char tpl[] = "abcXXXXXX";
    char suff_tpl[] = "abcXXXXXXsuff";
    char *filename = NULL;
    int fd;

    filename = tempnam("/var/tmp", "foo_");

    if (filename != NULL)
    {
        printf("generated tmp name (%s) in (%s:%s:%s)\n",
               filename, getenv("TMPDIR") ? getenv("TMPDIR") : "$TMPDIR",
               "/var/tmp", P_tmpdir);

        fd = open(filename, O_CREAT, S_IRWXU|S_IRUSR);
        if (fd != -1)
        {
            close(fd);
            unlink(filename);
            return 1;
        }
    }
    return 0;
}
```

In this example, Bug Finder flags `open` because it tries to use an unsecure temporary file. The file is opened without exclusive privileges. An attacker can access the file causing various risks on page 3-392.

Correction — Add `O_EXCL` Flag

One possible correction is to add the `O_EXCL` flag when you open the temporary file.

```
#define _BSD_SOURCE
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int test_temp()
{
    char tpl[] = "abcXXXXXX";
    char suff_tpl[] = "abcXXXXXXsuff";
    char *filename = NULL;
    int fd;

    filename = tempnam("/var/tmp", "foo_");

    if (filename != NULL)
    {
        printf("generated tmp name (%s) in (%s:%s:%s)\n",
               filename, getenv("TMPDIR") ? getenv("TMPDIR") : "$TMPDIR",
               "/var/tmp", P_tmpdir);
```

```
        fd = open(filename, O_CREAT|O_EXCL, S_IRWXU|S_IRUSR);
        if (fd != -1)
        {
            close(fd);
            unlink(filename);
            return 1;
        }
    }
    return 0;
}
```

Check Information

Group: Rec. 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FIO21-C

Introduced in R2019a

CERT C: Rec. FIO24-C

Do not open a file that is already open

Description

Rule Definition

Do not open a file that is already open.

Polyspace Implementation

This checker checks for **Opening previously opened resource**.

Examples

Opening previously opened resource

Issue

Opening previously opened resource checks for file opening functions that are opening an already opened file.

Risk

If you open a resource multiple times, you can encounter:

- A race condition when accessing the file.
- Undefined or unexpected behavior for that file.
- Portability issues when you run your program on different targets.

Fix

Once a resource is open, close the resource before reopening.

Example - File Reopened With New Permissions

```
#include <stdio.h>
const char* logfile = "my_file.log";

void doubleresourceopen()
{
    FILE* fpa = fopen(logfile, "w");
    if (fpa == NULL) {
        return;
    }
    (void)fprintf(fpa, "Writing");
    FILE* fpb = fopen(logfile, "r");
    (void)fclose(fpa);
    (void)fclose(fpb);
}
```

In this example, a `logfile` is opened in the first line of this function with write privileges. Halfway through the function, the `logfile` is opened again with read privileges.

Correction — Close Before Reopening

One possible correction is to close the file before reopening the file with different privileges.

```
#include <stdio.h>
const char* logfile = "my_file.log";

void doubleresourceopen()
{
    FILE* fpa = fopen(logfile, "w");
    if (fpa == NULL) {
        return;
    }
    (void)fprintf(fpa, "Writing");
    (void)fclose(fpa);
    FILE* fpb = fopen(logfile, "r");
    (void)fclose(fpb);
}
```

Check Information

Group: Rec. 09. Input Output (FIO)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

FIO24-C

Introduced in R2019a

CERT C: Rec. ENV01-C

Do not make assumptions about the size of an environment variable

Description

Rule Definition

Do not make assumptions about the size of an environment variable.

Polyspace Implementation

This checker checks for **Tainted NULL or non-null-terminated string**.

Examples

Tainted NULL or non-null-terminated string

Issue

Tainted NULL or non-null-terminated string looks for strings from unsecure sources that are being used in string manipulation routines that implicitly dereference the string buffer. For example, `strcpy` or `sprintf`.

Tainted NULL or non-null-terminated string raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

Note If you reference a string using the form `ptr[i]`, `*ptr`, or pointer arithmetic, Bug Finder raises a **Use of tainted pointer** defect instead. The **Tainted NULL or non-null-terminated string** defect is raised only when the pointer is used as a string.

Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

Example - Getting String from Input Argument

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

Correction — Validate the Data

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

int sanitize_str(char* s) {
  int res = 0;
  if (s && (strlen(s) > 0)) { // TAINTED_STRING only flagged here
    // - string is not null
    // - string has a positive and limited size
    // - TAINTED_STRING on strlen used as a firewall
    res = 1;
  }
  return res;
}

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
      strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
```

```
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  if (sanitize_str(userstr))
    strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}
```

Correction — Validate the Data

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

Check Information

Group: Rec. 10. Environment (ENV)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites
ENV01-C

Introduced in R2019a

CERT C: Rec. ERR00-C

Adopt and implement a consistent and comprehensive error-handling policy

Description

Rule Definition

Adopt and implement a consistent and comprehensive error-handling policy.

Polyspace Implementation

This checker checks for **Returned value of a sensitive function not checked**.

Examples

Returned value of a sensitive function not checked

Issue

This issue occurs when you call sensitive standard functions, but you:

- Ignore the return value.
- Use an output or a return value without testing the validity of the return value.

For this defect, two type of functions are considered: ***sensitive*** and ***critical sensitive***.

A ***sensitive*** function is a standard function that can encounter:

- Exhausted system resources (for example, when allocating resources)
- Changed privileges or permissions
- Tainted sources when reading, writing, or converting data from external sources
- Unsupported features despite an existing API

A ***critical sensitive*** function is a sensitive function that performs one of these critical or vulnerable tasks:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

Risk

If you do not check the return value of functions that perform sensitive or critical sensitive tasks, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

Fix

Before continuing with the program, test the return value of *critical sensitive* functions.

For *sensitive functions*, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

Example – Sensitive Function Return Ignored

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    pthread_attr_init(&attr);
}
```

This example shows a call to the sensitive function `pthread_attr_init`. The return value of `pthread_attr_init` is ignored, causing a defect.

Correction 1 – Cast Function to `(void)`

One possible correction is to cast the function to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of the sensitive function.

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    (void)pthread_attr_init(&attr);
}
```

Correction 2 – Test Return Value

One possible correction is to test the return value of `pthread_attr_init` to check for errors.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

void initialize() {
    pthread_attr_t attr;
    int result;

    result = pthread_attr_init(&attr);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

Example – Critical Function Return Ignored

```
#include <pthread.h>
extern void *start_routine(void *);

void returnnotchecked() {
```

```
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0));
    pthread_join(thread_id,  &res);
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), Polyspace does not ignore this *Return value of a sensitive function not checked* defect. The other critical function, `pthread_join`, returns value that is ignored implicitly. `pthread_join` uses the return value of `pthread_create`, which was not checked.

Correction — Test the Return Value of Critical Functions

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

Check Information

Group: Rec. 12. Error Handling (ERR)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites
ERR00-C

Introduced in R2019a

CERT C: Rec. API04-C

Provide a consistent and usable error-checking mechanism

Description

Rule Definition

Provide a consistent and usable error-checking mechanism.

Polyspace Implementation

This checker checks for **Returned value of a sensitive function not checked**.

Examples

Returned value of a sensitive function not checked

Issue

This issue occurs when you call sensitive standard functions, but you:

- Ignore the return value.
- Use an output or a return value without testing the validity of the return value.

For this defect, two type of functions are considered: ***sensitive*** and ***critical sensitive***.

A ***sensitive*** function is a standard function that can encounter:

- Exhausted system resources (for example, when allocating resources)
- Changed privileges or permissions
- Tainted sources when reading, writing, or converting data from external sources
- Unsupported features despite an existing API

A ***critical sensitive*** function is a sensitive function that performs one of these critical or vulnerable tasks:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

Risk

If you do not check the return value of functions that perform sensitive or critical sensitive tasks, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

Fix

Before continuing with the program, test the return value of *critical sensitive* functions.

For *sensitive functions*, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

Example – Sensitive Function Return Ignored

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    pthread_attr_init(&attr);
}
```

This example shows a call to the sensitive function `pthread_attr_init`. The return value of `pthread_attr_init` is ignored, causing a defect.

Correction 1 – Cast Function to `(void)`

One possible correction is to cast the function to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of the sensitive function.

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    (void)pthread_attr_init(&attr);
}
```

Correction 2 – Test Return Value

One possible correction is to test the return value of `pthread_attr_init` to check for errors.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

void initialize() {
    pthread_attr_t attr;
    int result;

    result = pthread_attr_init(&attr);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

Example – Critical Function Return Ignored

```
#include <pthread.h>
extern void *start_routine(void *);

void returnnotchecked() {
```

```
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0));
    pthread_join(thread_id,  &res);
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), Polyspace does not ignore this *Return value of a sensitive function not checked* defect. The other critical function, `pthread_join`, returns value that is ignored implicitly. `pthread_join` uses the return value of `pthread_create`, which was not checked.

Correction — Test the Return Value of Critical Functions

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

Check Information

Group: Rec. 13. Application Programming Interfaces (API)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites
API04-C

Introduced in R2019a

CERT C: Rec. CON01-C

Acquire and release synchronization primitives in the same module, at the same level of abstraction

Description

Rule Definition

Acquire and release synchronization primitives in the same module, at the same level of abstraction.

Polyspace Implementation

This checker checks for these issues:

- **Missing lock**.
- **Missing unlock**.
- **Double lock**.
- **Double unlock**.

Examples

Missing lock

Issue

Missing lock occurs when a task calls an unlock function before calling the corresponding lock function.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait till `my_task` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Risk

A call to an unlock function without a corresponding lock function can indicate a coding error. For instance, perhaps the unlock function does not correspond to the lock function that begins the critical section.

Fix

The fix depends on the root cause of the defect. For instance, if the defect occurs because of a mismatch between lock and unlock function, check the lock-unlock function pair in your Polyspace analysis configuration and fix the mismatch.

See examples of fixes below. To avoid the issue, you can follow the practice of calling the lock and unlock functions in the same module at the same level of abstraction. For instance, in this example, `func` calls the lock and unlock function at the same level but `func2` does not.

```
void func() {
  my_lock();
  {
    ...
  }
  my_unlock();
}

void func2() {
  {
   my_lock();
   ...
  }
  my_unlock();
}
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Missing lock

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset(void)
{
  begin_critical_section();
  global_var = 0;
  end_critical_section();
}

void my_task(void)
{
  global_var += 1;
  end_critical_section();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	**Specification**	
Configure multitasking manually on page 1-104	☑	
Tasks on page 1-108	`my_task`, `reset`	
Critical section details on page 1-119	**Starting routine**	**Ending routine**
	`begin_critical_section`	`end_critical_section`

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points my_task,reset
```

```
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

The example has two entry points, `my_task` and `reset`. `my_task` calls `end_critical_section` before calling `begin_critical_section`.

Correction — Provide Lock

One possible correction is to call the lock function `begin_critical_section` before the instructions in the critical section.

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset(void)
{
  begin_critical_section();
  global_var = 0;
  end_critical_section();
}

void my_task(void)
{
    begin_critical_section();
    global_var += 1;
    end_critical_section();
}
```

Example - Lock in Condition

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset() {
    begin_critical_section();
    global_var=0;
    end_critical_section();
}

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      if(index%10==0) {
        begin_critical_section();
        global_var ++;
      }
      end_critical_section();
```

```
      index++;
    }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Specification	
Configure multitasking manually on page 1-104	☑	
Tasks on page 1-108	`my_task`, `reset`	
Critical section details on page 1-119	**Starting routine**	**Ending routine**
	`begin_critical_section`	`end_critical_section`

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points my_task,reset
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

The example has two entry points, `my_task` and `reset`.

In the `while` loop, `my_task` leaves a critical section through the call `end_critical_section();`. In an iteration of the `while` loop:

- If `my_task` enters the `if` condition branch, the critical section begins through a call to `begin_critical_section`.
- If `my_task` does not enter the `if` condition branch and leaves the `while` loop, the critical section does not begin. Therefore, a **Missing lock** defect occurs.
- If `my_task` does not enter the `if` condition branch and continues to the next iteration of the `while` loop, the unlock function `end_critical_section` is called again. A **Double unlock** defect occurs.

Because `numCycles` is a `volatile` variable, it can take any value. Any of the cases above are possible. Therefore, a **Missing lock** defect and a **Double unlock** defect appear on the call `end_critical_section`.

Missing unlock

Issue

Missing unlock occurs when:

- A task calls a lock function.
- The task ends without a call to an unlock function.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task, `my_task`, calls a lock function, `my_lock`, other tasks calling `my_lock` must wait until `my_task` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, before analysis, you must specify the multitasking options. On the **Configuration** pane, select **Multitasking**.

Risk

An unlock function ends a critical section so that other waiting tasks can enter the critical section. A missing unlock function can result in tasks blocked for an unnecessary length of time.

Fix

Identify the critical section of code, that is, the section that you want to be executed as an atomic block. At the end of this section, call the unlock function that corresponds to the lock function used at the beginning of the section.

There can be other reasons and corresponding fixes for the defect. Perhaps you called the incorrect unlock function. Check the lock-unlock function pair in your Polyspace analysis configuration and fix the mismatch.

See examples of fixes below. To avoid the issue, you can follow the practice of calling the lock and unlock functions in the same module at the same level of abstraction. For instance, in this example, `func` calls the lock and unlock function at the same level but `func2` does not.

```
void func() {
  my_lock();
  {
    ...
  }
  my_unlock();
}

void func2() {
  {
   my_lock();
   ...
  }
  my_unlock();
}
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Missing Unlock

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset()
{
    begin_critical_section();
    global_var = 0;
    end_critical_section();
}

void my_task(void)
{
    begin_critical_section();
```

```
    global_var += 1;
}
```

In this example, to emulate multitasking behavior, specify the following options:

Option	Specification	
Configure multitasking manually on page 1-104	☑	
Tasks on page 1-108	`my_task`, `reset`	
Critical section details on page 1-119	**Starting routine**	**Ending routine**
	`begin_critical_section`	`end_critical_section`

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points my_task,reset
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

The example has two entry points, `my_task` and `reset`. `my_task` enters a critical section through the call `begin_critical_section();`. `my_task` ends without calling `end_critical_section`.

Correction — Provide Unlock

One possible correction is to call the unlock function `end_critical_section` after the instructions in the critical section.

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset(void)
{
    begin_critical_section();
    global_var = 0;
    end_critical_section();
}

void my_task(void)
{
    begin_critical_section();
    global_var += 1;
    end_critical_section();
}
```

Example - Unlock in Condition

```
void begin_critical_section(void);
```

```
void end_critical_section(void);

int global_var;

void reset() {
    begin_critical_section();
    global_var=0;
    end_critical_section();
}

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      begin_critical_section();
      global_var ++;
      if(index%10==0) {
        global_var = 0;
        end_critical_section();
      }
      index++;
    }
}
```

In this example, to emulate multitasking behavior, specify the following options.

Option	Specification	
Configure multitasking manually on page 1-104	☑	
Tasks on page 1-108	`my_task`, `reset`	
Critical section details on page 1-119	**Starting routine**	**Ending routine**
	`begin_critical_section`	`end_critical_section`

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points my_task,reset
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

The example has two entry points, `my_task` and `reset`.

In the `while` loop, `my_task` enters a critical section through the call `begin_critical_section();`. In an iteration of the `while` loop:

- If `my_task` enters the `if` condition branch, the critical section ends through a call to `end_critical_section`.
- If `my_task` does not enter the `if` condition branch and leaves the `while` loop, the critical section does not end. Therefore, a **Missing unlock** defect occurs.
- If `my_task` does not enter the `if` condition branch and continues to the next iteration of the `while` loop, the lock function `begin_critical_section` is called again. A **Double lock** defect occurs.

Because `numCycles` is a `volatile` variable, it can take any value. Any of the cases above is possible. Therefore, a **Missing unlock** defect and a **Double lock** defect appear on the call `begin_critical_section`.

Correction — Place Unlock Outside Condition

One possible correction is to call the unlock function `end_critical_section` outside the `if` condition.

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset() {
    begin_critical_section();
    global_var=0;
    end_critical_section();
}

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      begin_critical_section();
      global_var ++;
      if(index%10==0) {
        global_var=0;
      }
      end_critical_section();
      index++;
    }
}
```

Correction — Place Unlock in Every Conditional Branch

Another possible correction is to call the unlock function `end_critical_section` in every branches of the `if` condition.

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset() {
    begin_critical_section();
    global_var=0;
    end_critical_section();
}

void my_task(void) {
```

```
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      begin_critical_section();
      global_var ++;
      if(index%10==0) {
        global_var=0;
        end_critical_section();
      }
      else
        end_critical_section();
      index++;
    }
}
```

Double lock

Issue

Double lock occurs when:

- A task calls a lock function `my_lock`.
- The task calls `my_lock` again before calling the corresponding unlock function.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task `task1` calls a lock function `lock`, other tasks calling `lock` must wait until `task` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Risk

A call to a lock function begins a critical section so that other tasks have to wait to enter the same critical section. If the same lock function is called again within the critical section, the task blocks itself.

Fix

The fix depends on the root cause of the defect. A double lock defect often indicates a coding error. Perhaps you omitted the call to an unlock function to end a previous critical section and started the next critical section. Perhaps you wanted to use a different lock function for the second critical section.

Identify each critical section of code, that is, the section that you want to be executed as an atomic block. Call a lock function at the beginning of the section. Within the critical section, make sure that you do not call the lock function again. At the end of the section, call the unlock function that corresponds to the lock function.

See examples of fixes below. To avoid the issue, you can follow the practice of calling the lock and unlock functions in the same module at the same level of abstraction. For instance, in this example, `func` calls the lock and unlock function at the same level but `func2` does not.

```
void func() {
  my_lock();
```

```
  {
    ...
  }
  my_unlock();
}

void func2() {
  {
   my_lock();
   ...
  }
  my_unlock();
}
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Double Lock

```
int global_var;

void lock(void);
void unlock(void);

void task1(void)
{
    lock();
    global_var += 1;
    lock();
    global_var += 1;
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	**Specification**	
Configure multitasking manually on page 1-104	☑	
Tasks on page 1-108	`my_task`, `reset`	
Critical section details on page 1-119	**Starting routine**	**Ending routine**
	`lock`	`unlock`

On the command-line, you can use the following:

```
polyspace-bug-finder
  -entry-points task1,task2
```

```
   -critical-section-begin lock:cs1
   -critical-section-end unlock:cs1
```

`task1` enters a critical section through the call `lock();`. `task1` calls `lock` again before it leaves the critical section through the call `unlock();`.

Correction — Remove First Lock

If you want the first `global_var+=1;` to be outside the critical section, one possible correction is to remove the first call to `lock`. However, if other tasks are using `global_var`, this code can produce a `Data race` error.

```
int global_var;

void lock(void);
void unlock(void);

void task1(void)
{
    global_var += 1;
    lock();
    global_var += 1;
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

Correction — Remove Second Lock

If you want the first `global_var+=1;` to be inside the critical section, one possible correction is to remove the second call to `lock`.

```
int global_var;

void lock(void);
void unlock(void);

void task1(void)
{
    lock();
    global_var += 1;
    global_var += 1;
    unlock();
}

void task2(void)
{
```

```
    lock();
    global_var += 1;
    unlock();
}
```

Correction — Add Another Unlock

If you want the second `global_var+=1;` to be inside a critical section, another possible correction is to add another call to `unlock`.

```
int global_var;

void lock(void);
void unlock(void);

void task1(void)
{
    lock();
    global_var += 1;
    unlock();
    lock();
    global_var += 1;
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

Example - Double Lock with Function Call

```
int global_var;

void lock(void);
void unlock(void);

void performOperation(void) {
  lock();
  global_var++;
}

void task1(void)
{
    lock();
    global_var += 1;
    performOperation();
    unlock();
}

void task2(void)
```

```
{
    lock();
    global_var += 1;
    unlock();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Specification	
Configure multitasking manually on page 1-104	☑	
Tasks on page 1-108	`my_task`, `reset`	
Critical section details on page 1-119	**Starting routine**	**Ending routine**
	`lock`	`unlock`

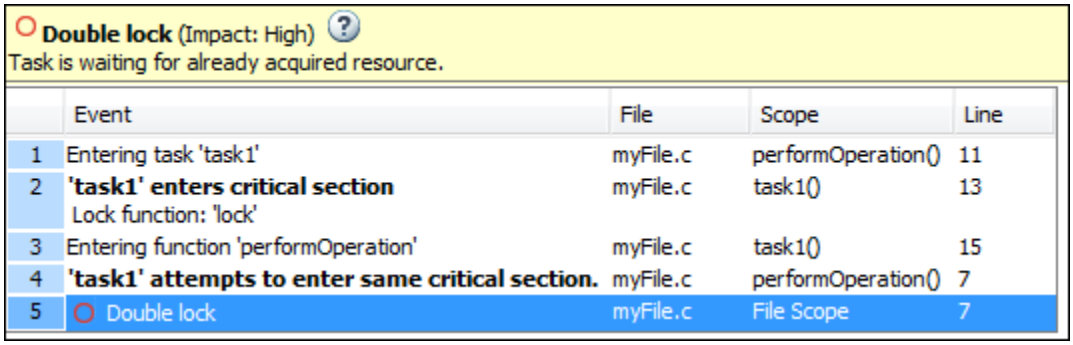
On the command-line, you can use the following:

```
polyspace-bug-finder
  -entry-points task1,task2
  -critical-section-begin lock:cs1
  -critical-section-end unlock:cs1
```

`task1` enters a critical section through the call `lock();`. `task1` calls the function `performOperation`. In `performOperation`, `lock` is called again even though `task1` has not left the critical section through the call `unlock();`.

In the result details for the defect, you see the sequence of instructions leading to the defect. For instance, you see that following the first entry into the critical section, the execution path:

- Enters function `performOperation`.
- Inside `performOperation`, attempts to enter the same critical section once again.



You can click each event to navigate to the corresponding line in the source code.

Correction — Remove Second Lock

One possible correction is to remove the call to `lock` in `task1`.

```
int global_var;
```

```
void lock(void);
void unlock(void);

void performOperation(void) {
  global_var++;
}

void task1(void)
{
    lock();
    global_var += 1;
    performOperation();
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

Double unlock

Issue

Double unlock occurs when:

- A task calls a lock function `my_lock`.
- The task calls the corresponding unlock function `my_unlock`.
- The task calls `my_unlock` again. The task does not call `my_lock` a second time between the two calls to `my_unlock`.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task `task1` calls a lock function `my_lock`, other tasks calling `my_lock` must wait until `task1` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Risk

A double unlock defect can indicate a coding error. Perhaps you wanted to call a different unlock function to end a different critical section. Perhaps you called the unlock function prematurely the first time and only the second call indicates the end of the critical section.

Fix

The fix depends on the root cause of the defect.

Identify each critical section of code, that is, the section that you want to be executed as an atomic block. Call a lock function at the beginning of the section. Only at the end of the section, call the unlock function that corresponds to the lock function. Remove any other redundant call to the unlock function.

See examples of fixes below. To avoid the issue, you can follow the practice of calling the lock and unlock functions in the same module at the same level of abstraction. For instance, in this example, `func` calls the lock and unlock function at the same level but `func2` does not.

```
void func() {
  my_lock();
  {
    ...
  }
  my_unlock();
}

void func2() {
  {
   my_lock();
   ...
  }
  my_unlock();
}
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Double Unlock

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void task1(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}

void task2(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Value
Configure multitasking manually on page 1-104	☑

Option	Value	
Tasks on page 1-108	task1 task2	
Critical section details on page 1-119	**Starting routine**	**Ending routine**
	BEGIN_CRITICAL_SECTION	END_CRITICAL_SECTION

On the command-line, you can use the following:

```
polyspace-bug-finder
  -entry-points task1,task2
  -critical-section-begin BEGIN_CRITICAL_SECTION:cs1
  -critical-section-end END_CRITICAL_SECTION:cs1
```

`task1` enters a critical section through the call `BEGIN_CRITICAL_SECTION();`. `task1` leaves the critical section through the call `END_CRITICAL_SECTION();`. `task1` calls `END_CRITICAL_SECTION` again without an intermediate call to `BEGIN_CRITICAL_SECTION`.

Correction — Remove Second Unlock

If you want the second `global_var+=1;` to be outside the critical section, one possible correction is to remove the second call to `END_CRITICAL_SECTION`. However, if other tasks are using `global_var`, this code can produce a `Data race` error.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void task1(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
    global_var += 1;
}

void task2(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

Correction — Remove First Unlock

If you want the second `global_var+=1;` to be inside the critical section, one possible correction is to remove the first call to `END_CRITICAL_SECTION`.

```
int global_var;
```

```
void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void task1(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    global_var += 1;
    END_CRITICAL_SECTION();
}

void task2(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

Correction — Add Another Lock

If you want the second `global_var+=1;` to be inside a critical section, another possible correction is to add another call to `BEGIN_CRITICAL_SECTION`.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void task1(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}

void task2(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

Check Information

Group: Rec. 14. Concurrency (CON)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites
CON01-C

Introduced in R2019a

CERT C: Rec. CON05-C

Do not perform operations that can block while holding a lock

Description

Rule Definition

Do not perform operations that can block while holding a lock.

Polyspace Implementation

This checker checks for **Blocking operation while holding lock**.

Examples

Blocking operation while holding lock

Issue

Blocking operation while holding lock occurs when a task (thread) performs a potentially lengthy operation while holding a lock.

The checker considers calls to these functions as potentially lengthy:

- Functions that access a network such as `recv`
- System call functions such as `fork`, `pipe` and `system`
- Functions for I/O operations such as `getchar` and `scanf`
- File handling functions such as `fopen`, `remove` and `lstat`
- Directory manipulation functions such as `mkdir` and `rmdir`

The checker automatically detects certain primitives that hold and release a lock, for instance, `pthread_mutex_lock` and `pthread_mutex_unlock`. For the full list of primitives that are automatically detected, see “Auto-Detection of Thread Creation and Critical Section in Polyspace”.

Risk

If a thread performs a lengthy operation when holding a lock, other threads that use the lock have to wait for the lock to be available. As a result, system performance can slow down or deadlocks can occur.

Fix

Perform the blocking operation before holding the lock or after releasing the lock.

Some functions detected by this checker can be called in a way that does not make them potentially lengthy. For instance, the function `recv` can be called with the parameter `O_NONBLOCK` which causes the call to fail if no message is available. When called with this parameter, `recv` does not wait for a message to become available.

Example - Network I/O Operations with `recv` While Holding Lock

```
#include <pthread.h>
#include <sys/socket.h>
```

```
pthread_mutexattr_t attr;
pthread_mutex_t mutex;

void thread_foo(void *ptr) {
  unsigned int num;
  int result;
  int sock;

  /* sock is a connected TCP socket */

  if ((result = pthread_mutex_lock(&mutex)) != 0) {
    /* Handle Error */
  }

  if ((result = recv(sock, (void *)&num, sizeof(unsigned int), 0)) < 0) {
    /* Handle Error */
  }

  /* ... */

  if ((result = pthread_mutex_unlock(&mutex)) != 0) {
    /* Handle Error */
  }
}

int main() {
  pthread_t thread;
  int result;

  if ((result = pthread_mutexattr_settype(
      &attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_init(&mutex, &attr)) != 0) {
    /* Handle Error */
  }

  if (pthread_create(&thread, NULL,(void*(*)(void*))& thread_foo, NULL) != 0) {
    /* Handle Error */
  }

  /* ... */

  pthread_join(thread, NULL);

  if ((result = pthread_mutex_destroy(&mutex)) != 0) {
    /* Handle Error */
  }

  return 0;
}
```

In this example, in each thread created with `pthread_create`, the function `thread_foo` performs a network I/O operation with `recv` after acquiring a lock with `pthread_mutex_lock`. Other threads

using the same lock variable `mutex` have to wait for the operation to complete and the lock to become available.

Correction — Perform Blocking Operation Before Acquiring Lock

One possible correction is to call `recv` before acquiring the lock.

```
#include <pthread.h>
#include <sys/socket.h>

pthread_mutexattr_t attr;
pthread_mutex_t mutex;

void thread_foo(void *ptr) {
  unsigned int num;
  int result;
  int sock;

  /* sock is a connected TCP socket */
  if ((result = recv(sock, (void *)&num, sizeof(unsigned int), 0)) < 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_lock(&mutex)) != 0) {
    /* Handle Error */
  }

    /* ... */

  if ((result = pthread_mutex_unlock(&mutex)) != 0) {
    /* Handle Error */
  }
}

int main() {
  pthread_t thread;
  int result;

  if ((result = pthread_mutexattr_settype(
      &attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_init(&mutex, &attr)) != 0) {
    /* Handle Error */
  }

  if (pthread_create(&thread, NULL,(void*(*)(void*))& thread_foo, NULL) != 0) {
    /* Handle Error */
  }

  /* ... */

  pthread_join(thread, NULL);

  if ((result = pthread_mutex_destroy(&mutex)) != 0) {
    /* Handle Error */
  }
```

```
  return 0;
}
```

Check Information

Group: Rec. 14. Concurrency (CON)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

CON05-C

Introduced in R2019a

CERT C: Rec. MSC01-C

Strive for logical completeness

Description

Rule Definition

Strive for logical completeness.

Polyspace Implementation

This checker checks for **Missing case for switch condition**.

Examples

Missing case for switch condition

Issue

Missing case for switch condition occurs when the `switch` variable can take values that are not covered by a `case` statement.

Note Bug Finder only raises a defect if the switch variable is not full range.

Risk

If the `switch` variable takes a value that is not covered by a `case` statement, your program can have unintended behavior.

A switch-statement that makes a security decision is particularly vulnerable when all possible values are not explicitly handled. An attacker can use this situation to deviate the normal execution flow.

Fix

It is good practice to use a `default` statement as a catch-all for values that are not covered by a `case` statement. Even if the `switch` variable takes an unintended value, the resulting behavior can be anticipated.

Example - Missing Default Condition

```
#include <stdio.h>
#include <string.h>

typedef enum E
{
    ADMIN=1,
    GUEST,
    UNKNOWN = 0
} LOGIN;

static LOGIN system_access(const char *username) {
```

```
  LOGIN user = UNKNOWN;

  if ( strcmp(username, "root") == 0 )
    user = ADMIN;

  if ( strcmp(username, "friend") == 0 )
    user = GUEST;

  return user;
}

int identify_bad_user(const char * username)
{
    int r=0;

    switch( system_access(username) )
    {
    case ADMIN:
        r = 1;
        break;
    case GUEST:
        r = 2;
    }

    printf("Welcome!\n");
    return r;
}
```

In this example, the `enum` parameter `User` can take a value `UNKNOWN` that is not covered by a `case` statement.

Correction — Add a Default Condition

One possible correction is to add a default condition for possible values that are not covered by a `case` statement.

```
#include <stdio.h>
#include <string.h>

typedef enum E
{
    ADMIN=1,
    GUEST,
    UNKNOWN = 0
} LOGIN;

static LOGIN system_access(const char *username) {
  LOGIN user = UNKNOWN;

  if ( strcmp(username, "root") == 0 )
    user = ADMIN;

  if ( strcmp(username, "friend") == 0 )
    user = GUEST;

  return user;
}
```

```
int identify_bad_user(const char * username)
{
    int r=0;

    switch( system_access(username) )
    {
    case ADMIN:
        r = 1;
        break;
    case GUEST:
        r = 2;
    break;
    default:
        printf("Invalid login credentials!\n");
    }

    printf("Welcome!\n");
    return r;
}
```

Check Information

Group: Rec. 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MSC01-C

Introduced in R2019a

CERT C: Rec. MSC04-C

Use comments consistently and in a readable fashion

Description

Rule Definition

Use comments consistently and in a readable fashion.

Polyspace Implementation

This checker checks for **Use of /* and // within a comment**.

Examples

Use of /* and // within a comment

Issue

The issue occurs when you use the character sequences `/*` and `//` within a comment.

You cannot annotate this rule in the source code. For information on annotations, see “Annotate Code and Hide Known or Acceptable Results”.

Risk

These character sequences are not allowed in code comments because:

- If your code contains a `/*` or a `//` in a `/* */` comment, it typically means that you have inadvertently commented out code.
- If your code contains a `/*` in a `//` comment, it typically means that you have inadvertently uncommented a `/* */` comment.

Example - /* Used in // Comments

```
int x;
int y;
int z;

void non_compliant_comments ( void )
{
    x = y //      /* Non-compliant
        + z
        //  */
        ;
    z++;    //    Compliant with exception: // permitted within a // comment
}

void compliant_comments ( void )
{
    x = y /*      Compliant
      + z
      */
```

```
        ;
    z++;    //    Compliant with exception: // is permitted within a // comment
}
```

In this example, in the `non_compliant_comments` function, the `/*` character occurs in what appears to be a `//` comment, violating the rule. Because of the comment structure, the operation that takes place is `x = y + z;`. However, without the two `//`-s, an entirely different operation `x=y;` takes place. It is not clear which operation is intended.

Use a comment format that makes your intention clear. For instance, in the `compliant_comments` function, it is clear that the operation `x=y;` is intended.

Check Information

Group: Rec. 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MSC04-C

Introduced in R2019a

CERT C: Rec. MSC12-C

Detect and remove code that has no effect or is never executed

Description

Rule Definition

Detect and remove code that has no effect or is never executed.

Polyspace Implementation

This checker checks for these issues:

- **Unreachable code**.
- **Dead code**.
- **Useless if**.
- **Write without a further read**.

Examples

Unreachable code

Issue

The issue occurs when your project contains code that is unreachable.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

The Code Prover run-time check for unreachable code shows more cases than the MISRA checker for this rule. See also `Unreachable code`. The run-time check performs a more exhaustive analysis. In the process, the check can show some instances that are not strictly unreachable code but unreachable only in the context of the analysis. For instance, in the following code, the run-time check shows a potential division by zero in the first line and then removes the zero value of `flag` for the rest of the analysis. Therefore, it considers the `if` block unreachable.

```
val=1.0/flag;
if(!flag) {}
```

The MISRA checker is designed to prevent these kinds of results.

Risk

Unless a program exhibits any undefined behavior, unreachable code cannot execute. The unreachable code cannot affect the program output. The presence of unreachable code can indicate an error in the program logic. Unreachable code that the compiler does not remove wastes resources, for example:

- It occupies space in the target machine memory.
- Its presence can cause a compiler to select longer, slower jump instructions when transferring control around the unreachable code.

- Within a loop, it can prevent the entire loop from residing in an instruction cache.

Example - Code Following `return` Statement

```
enum light { red, amber, red_amber, green };

enum light next_light ( enum light color )
{
    enum light res;

    switch ( color )
    {
    case red:
        res = red_amber;
        break;
    case red_amber:
        res = green;
        break;
    case green:
        res = amber;
        break;
    case amber:
        res = red;
        break;
    default:
    {
        error_handler ();
        break;
    }
    }

    res = color;
    return res;
    res = color;     /* Non-compliant */
}
```

In this example, the rule is violated because there is an unreachable operation following the `return` statement.

Dead code

Issue

The issue occurs when the analysis detects a reachable operation that does not affect program behavior if the operation is removed.

Polyspace Bug Finder detects useless write operations during analysis.

Polyspace Code Prover does not detect useless write operations. For instance, if you assign a value to a local variable but do not read it later, Polyspace Code Prover does not detect this useless assignment. Use Polyspace Bug Finder to detect such useless write operations. For more information, see MISRA C:2012 in Polyspace Bug Finder on page 5-133.

In Code Prover, you can also see a difference in results based on your choice for the option `Verification level (-to)`. See “Check for Coding Standard Violations”.

Risk

If an operation is reachable but removing the operation does not affect program behavior, the operation constitutes dead code.

The presence of dead code can indicate an error in the program logic. Because a compiler can remove dead code, its presence can cause confusion for code reviewers.

Operations involving language extensions such as `__asm ( "NOP" );` are not considered dead code.

Example - Redundant Operations

```
extern volatile unsigned int v;
extern char *p;

void f ( void ) {
    unsigned int x;


    ( void ) v;      /* Compliant - Exception*/
    ( int ) v;       /* Non-compliant  */
    v >> 3;          /* Non-compliant  */

    x = 3;           /* Non-compliant - Detected in Bug Finder only */

    *p++;            /* Non-compliant  */
    ( *p )++;        /* Compliant  */
}
```

In this example, the rule is violated when an operation is performed on a variable, but the result of that operation is not used. For instance,

- The operations `(int)` and `>>` on the variable `v` are redundant because the results are not used.
- The operation `=` is redundant because the local variable `x` is not read after the operation.
- The operation `*` on `p++` is redundant because the result is not used.

The rule is not violated when:

- A variable is cast to `void`. The cast indicates that you are intentionally not using the value.
- The result of an operation is used. For instance, the operation `*` on `p` is not redundant, because `*p` is incremented.

Example - Redundant Function Call

```
void g ( void ) {
               /* Compliant  */
}

void h ( void) {
     g( );     /* Non-compliant */
}
```

In this example, `g` is an empty function. Though the function itself does not violate the rule, a call to the function violates the rule.

Useless if

Issue

This issue occurs on `if`-statements where the condition is always true. This defect occurs only on if-statements that do not have an else-statement.

This defect shows unnecessary `if`-statements when there is no difference in code execution if the `if`-statement is removed.

Risk

Unnecessary `if` statements often indicate a coding error. Perhaps the `if` condition is coded incorrectly or the `if` statement is not required at all.

Fix

The fix depends on the root cause of the defect. For instance, the root cause can be an error condition that is checked twice on the same execution path, making the second check redundant.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If the redundant condition represents defensive coding practices and you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example – `if` with Enumerated Type

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    if (card < 7) {
        do_something(card);
    }
}
```

The type `suit` is enumerated with five options. However, the conditional expression `card < 7` always evaluates to true because `card` can be at most 5. The `if` statement is unnecessary.

Correction 1 — Change Condition

One possible correction is to change the if-condition in the code. In this correction, the 7 is changed to `UNKNOWN_SUIT` to relate directly to the type of `card`.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
```

```
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    if (card > UNKNOWN_SUIT) {
        do_something(card);
    }
}
```

Correction — Remove If

Another possible correction is to remove the if-condition in the code. Because the condition is always true, you can remove the condition to simplify your code.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    do_something(card);
}
```

Write without a further read

Issue

This issue occurs when a value assigned to a variable is never read.

For instance, you write a value to a variable and then write a second value before reading the previous value. The first write operation is redundant.

Risk

Redundant write operations often indicate programming errors. For instance, you forgot to read the variable between two successive write operations or unintentionally read a different variable.

Fix

Identify the reason why you write to the variable but do not read it later. Look for common programming errors such as accidentally reading a different variable with a similar name.

If you determine that the write operation is redundant, remove the operation.

Example – Write Without Further Read Error

```
void sensor_amplification(void)
{
    extern int getsensor(void);
```

```
    int level;

    level = 4 * getsensor();
    /* Defect: Useless write */
}
```

After the variable `level` gets assigned the value `4 * getsensor()`, it is not read.

Correction — Use Value After Assignment

One possible correction is to use the variable `level` after the assignment.

```
#include <stdio.h>

void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();

    /* Fix: Use level after assignment */
    printf("The value is %d", level);

}
```

The variable `level` is printed, reading the new value.

Check Information

Group: Rec. 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MSC12-C

Introduced in R2019a

CERT C: Rec. MSC13-C

Detect and remove unused values

Description

Rule Definition

Detect and remove unused values.

Polyspace Implementation

This checker checks for these issues:

- **Unused parameter**.
- **Write without a further read**.

Examples

Unused parameter

Issue

Unused parameter occurs when a function parameter is neither read nor written in the function body.

Risk

Unused function parameters cause the following issues:

- Indicate that the code is possibly incomplete. The parameter is possibly intended for an operation that you forgot to code.
- If the copied objects are large, redundant copies can slow down performance.

Fix

Determine if you intend to use the parameters. Otherwise, remove parameters that you do not use in the function body.

You can intentionally have unused parameters. For instance, you have parameters that you intend to use later when you add enhancements to the function. Add a code comment indicating your intention for later use. The code comment helps you or a code reviewer understand why your function has unused parameters.

Alternatively, add a statement such as `(void)var;` in the function body. `var` is the unused parameter. You can define a macro that expands to this statement and add the macro to the function body.

Example - Unused Parameter

```
void func(int* xptr, int* yptr, int flag) {
    if(flag==1) {
        *xptr=0;
```

```
    }
    else {
        *xptr=1;
    }
}

int main() {
    int x,y;
    func(&x,&y,1);
    return 0;
}
```

In this example, the parameter `yptr` is not used in the body of `func`.

Correction — Use Parameter

One possible correction is to check if you intended to use the parameter. Fix your code if you intended to use the parameter.

```
void func(int* xptr, int* yptr, int flag) {
    if(flag==1) {
        *xptr=0;
        *yptr=1;
    }
    else {
        *xptr=1;
        *yptr=0;
    }
}

int main() {
    int x,y;
    func(&x,&y,1);
    return 0;
}
```

Correction — Explicitly Indicate Unused Parameter

Another possible correction is to explicitly indicate that you are aware of the unused parameter.

```
#define UNUSED(x) (void)x

void func(int* xptr, int* yptr, int flag) {
    UNUSED(yptr);
    if(flag==1) {
        *xptr=0;
    }
    else {
        *xptr=1;
    }
}

int main() {
    int x,y;
    func(&x,&y,1);
    return 0;
}
```

Write without a further read

Issue

Write without a further read occurs when a value assigned to a variable is never read.

For instance, you write a value to a variable and then write a second value before reading the previous value. The first write operation is redundant.

Risk

Redundant write operations often indicate programming errors. For instance, you forgot to read the variable between two successive write operations or unintentionally read a different variable.

Fix

Identify the reason why you write to the variable but do not read it later. Look for common programming errors such as accidentally reading a different variable with a similar name.

If you determine that the write operation is redundant, remove the operation.

Example - Write Without Further Read Error

```
void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();
    /* Defect: Useless write */
}
```

After the variable `level` gets assigned the value `4 * getsensor()`, it is not read.

Correction — Use Value After Assignment

One possible correction is to use the variable `level` after the assignment.

```
#include <stdio.h>

void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();

    /* Fix: Use level after assignment */
    printf("The value is %d", level);

}
```

The variable `level` is printed, reading the new value.

Check Information

Group: Rec. 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MSC13-C

Introduced in R2019a

CERT C: Rec. MSC15-C

Do not depend on undefined behavior

Description

Rule Definition

Do not depend on undefined behavior.

Polyspace Implementation

This checker checks for **Undefined behavior**.

Examples

Undefined behavior

Issue

The issue occurs when the analysis detects undefined or critical unspecified behaviour.

Check Information

Group: Rec. 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics
“Check for Coding Standard Violations”

External Websites
MSC15-C

Introduced in R2019a

CERT C: Rec. MSC17-C

Finish every set of statements associated with a case label with a break statement

Description

Rule Definition

Finish every set of statements associated with a case label with a break statement.

Polyspace Implementation

This checker checks for **Missing break of switch case**.

Examples

Missing break of switch case

Issue

Missing break of switch case looks for switch cases that do not end in a `break` statement. If the case does not have a code comment after it, Polyspace assumes the missing break is not intentional and raises a defect.

Risk

Switch cases without break statements fall through to the next switch case. If this fall-through is not intended, the switch case can unintentionally execute code and end the switch with unexpected results.

Fix

If you do not want a break for the highlighted switch case, add a comment to your code to document why this case falls through to the next case. This comment removes the defect from your results and makes your code more maintainable.

If you forgot the break, add it before the end of the switch case.

Example - Switch Without Break Statements

```
enum WidgetEnum { WE_W, WE_X, WE_Y, WE_Z } widget_type;

extern void demo_do_something_for_WE_W(void);
extern void demo_do_something_for_WE_X(void);
extern void demo_report_error(void);

void bug_missingswitchbreak(enum WidgetEnum wt)
{
    /*
      In this non-compliant code example, the case where widget_type is WE_W lacks a
      break statement. Consequently, statements that should be executed only when
      widget_type is WE_X are executed even when widget_type is WE_W.
    */
    switch (wt)
    {
```

```
      case WE_W:
        demo_do_something_for_WE_W();
      case WE_X:
        demo_do_something_for_WE_X();
      default:
        /* Handle error condition */
        demo_report_error();
    }
}
```

In this example, there are two cases without `break` statements. When `wt` is `WE_W`, the statements for `WE_W`, `WE_X`, and the `default` case execute because the program falls through the two cases without a break. No defect is raised on the `default` case or last case because it does not need a break statement.

Correction — Add a Comment or `break`

To fix this example, either add a comment to mark and document the acceptable fall-through or add a break statement to avoid fall-through. In this example, case `WE_W` is supposed to fall through, so a comment is added to explicitly state this action. For the second case, a break statement is added to avoid falling through to the `default` case.

```
enum WidgetEnum { WE_W, WE_X, WE_Y, WE_Z } widget_type;

extern void demo_do_something_for_WE_W(void);
extern void demo_do_something_for_WE_X(void);
extern void demo_report_error(void);

void corrected_missingswitchbreak(enum WidgetEnum wt)
{
    switch (wt)
    {
      case WE_W:
        demo_do_something_for_WE_W();
        /* fall through to WE_X*/
      case WE_X:
        demo_do_something_for_WE_X();
        break;
      default:
        /* Handle error condition */
        demo_report_error();
    }
}
```

Check Information

Group: Rec. 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MSC17-C

Introduced in R2019a

CERT C: Rec. MSC18-C

Be careful while handling sensitive data, such as passwords, in program code

Description

Rule Definition

Be careful while handling sensitive data, such as passwords, in program code.

Polyspace Implementation

This checker checks for these issues:

- **Constant block cipher initialization vector**.
- **Constant cipher key**.
- **Predictable block cipher initialization vector**.
- **Predictable cipher key**.
- **Sensitive heap memory not cleared before release**.
- **Uncleared sensitive data in stack**.
- **Unsafe standard encryption function**.

Examples

Constant block cipher initialization vector

Issue

Constant block cipher initialization vector occurs when you use a constant for the initialization vector (IV) during encryption.

Risk

Using a constant IV is equivalent to not using an IV. Your encrypted data is vulnerable to dictionary attacks.

Block ciphers break your data into blocks of fixed size. Block cipher modes such as CBC (Cipher Block Chaining) protect against dictionary attacks by XOR-ing each block with the encrypted output from the previous block. To protect the first block, these modes use a random initialization vector (IV). If you use a constant IV to encrypt multiple data streams that have a common beginning, your data becomes vulnerable to dictionary attacks.

Fix

Produce a random IV by using a strong random number generator.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

Example - Constants Used for Initialization Vector

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

/* Using the cryptographic routines */

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16] = {'1', '2', '3', '4','5','6','b','8','9',
                                '1','2','3','4','5','6','7'};
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

In this example, the initialization vector `iv` has constants only. The constant initialization vector makes your cipher vulnerable to dictionary attacks.

Correction — Use Random Initialization Vector

One possible correction is to use a strong random number generator to produce the initialization vector. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

/* Using the cryptographic routines */

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16];
    RAND_bytes(iv, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

Constant cipher key

Issue

Constant cipher key occurs when you use a constant for the encryption or decryption key.

Risk

If you use a constant for the encryption or decryption key, an attacker can retrieve your key easily.

You use a key to encrypt and later decrypt your data. If a key is easily retrieved, data encrypted using that key is not secure.

Fix

Produce a random key by using a strong random number generator.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

Example - Constants Used for Key

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16] = {'1', '2', '3', '4','5','6','b','8','9',
                                 '1','2','3','4','5','6','7'};
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

In this example, the cipher key, `key`, has constants only. An attacker can easily retrieve a constant key.

Correction — Use Random Key

Use a strong random number generator to produce the cipher key. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16];
    RAND_bytes(key, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

Predictable block cipher initialization vector

Issue

Predictable block cipher initialization vector occurs when you use a weak random number generator for the block cipher initialization vector.

Risk

If you use a weak random number generator for the initiation vector, your data is vulnerable to dictionary attacks.

Block ciphers break your data into blocks of fixed size. Block cipher modes such as CBC (Cipher Block Chaining) protect against dictionary attacks by XOR-ing each block with the encrypted output from the previous block. To protect the first block, these modes use a random initialization vector (IV). If you use a weak random number generator for your IV, your data becomes vulnerable to dictionary attacks.

Fix

Use a strong pseudo-random number generator (PRNG) for the initialization vector. For instance, use:

- OS-level PRNG such as `/dev/random` on UNIX or `CryptGenRandom()` on Windows
- Application-level PRNG such as Advanced Encryption Standard (AES) in Counter (CTR) mode, HMAC-SHA1, etc.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

Example - Predictable Initialization Vector

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16];
    RAND_pseudo_bytes(iv, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

In this example, the function `RAND_pseudo_bytes` declared in `openssl/rand.h` produces the initialization vector. The byte sequences that `RAND_pseudo_bytes` generates are not necessarily unpredictable.

Correction — Use Strong Random Number Generator

Use a strong random number generator to produce the initialization vector. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16];
    RAND_bytes(iv, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

Predictable cipher key

Issue

Predictable cipher key occurs when you use a weak random number generator for the encryption or decryption key.

Risk

If you use a weak random number generator for the encryption or decryption key, an attacker can retrieve your key easily.

You use a key to encrypt and later decrypt your data. If a key is easily retrieved, data encrypted using that key is not secure.

Fix

Use a strong pseudo-random number generator (PRNG) for the key. For instance:

- Use an OS-level PRNG such as `/dev/random` on UNIX or `CryptGenRandom()` on Windows

- Use an application-level PRNG such as Advanced Encryption Standard (AES) in Counter (CTR) mode, HMAC-SHA1, etc.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

Example - Predictable Cipher Key

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16];
    RAND_pseudo_bytes(key, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

In this example, the function `RAND_pseudo_bytes` declared in `openssl/rand.h` produces the cipher key. However, the byte sequences that `RAND_pseudo_bytes` generates are not necessarily unpredictable.

Correction — Use Strong Random Number Generator

One possible correction is to use a strong random number generator to produce the cipher key. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16];
    RAND_bytes(key, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

Sensitive heap memory not cleared before release

Issue

Sensitive heap memory not cleared before release detects dynamically allocated memory containing sensitive data. If you do not clear the sensitive data when you free the memory, Bug Finder raises a defect on the `free` function.

Risk

If the memory zone is reallocated, an attacker can still inspect the sensitive data in the old memory zone.

Fix

Before calling `free`, clear out the sensitive data using `memset` or `SecureZeroMemory`.

Example - Sensitive Buffer Freed, Not Cleared

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>

void sensitiveheapnotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char* buf = (char*) malloc(1024);
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    free(buf);
}
```

In this example, the function uses a buffer of passwords and frees the memory before the end of the function. However, the data in the memory is not cleared by using the `free` command.

Correction — Nullify Data

One possible correction is to write over the data to clear out the sensitive information. This example uses `memset` to write over the data with zeros.

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>
#include <assert.h>

#define isNull(arr) for(int i=0;i<(sizeof(arr)/sizeof(arr[0]));i++) assert(arr[i]==0)

void sensitiveheapnotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char* buf = (char*) malloc(1024);

    if (buf) {
        getpwnam_r(my_user, &pwd, buf, bufsize, &result);
        memset(buf, 0, (size_t)1024);
        isNull(buf);
        free(buf);
    }
}
```

Uncleared sensitive data in stack

Issue

Uncleared sensitive data in stack detects static memory containing sensitive data. If you do not clear the sensitive data from your stack before exiting the function or program, Bug Finder raises a defect on the last curly brace.

Risk

Leaving sensitive information in your stack, such as passwords or user information, allows an attacker additional access to the information after your program has ended.

Fix

Before exiting a function or program, clear out the memory zones that contain sensitive data by using `memset` or `SecureZeroMemory`.

Example - Static Buffer of Password Information

```
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>

void bug_sensitivestacknotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
}
```

In this example, a static buffer is filled with password information. The program frees the stack memory at the end of the program. However, the data is still accessible from the memory.

Correction — Clear Memory

One possible correction is to write over the memory before exiting the function. This example uses `memset` to clear the data from the buffer memory.

```
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>
#include <assert.h>

#define isNull(arr) for(int i=0; i<(sizeof(arr)/sizeof(arr[0])); i++) assert(arr[i]==0)

void corrected_sensitivestacknotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    memset(buf, 0, (size_t)1024);
    isNull(buf);
}
```

Unsafe standard encryption function

Issue

Unsafe standard encryption function detects use of functions with a broken or weak cryptographic algorithm. For example, `crypt` is not reentrant and is based on the risky Data Encryption Standard (DES).

Risk

The use of a broken, weak, or nonstandard algorithm can expose sensitive information to an attacker. A determined hacker can access the protected data using various techniques.

If the weak function is nonreentrant, when you use the function in concurrent programs, there is an additional race condition risk.

Fix

Avoid functions that use these encryption algorithms. Instead, use a reentrant function that uses a stronger encryption algorithm.

Note Some implementations of `crypt` support additional, possibly more secure, encryption algorithms.

Example - Decrypting Password Using `crypt`

```
#define _GNU_SOURCE
#include <pwd.h>
#include <string.h>
#include <crypt.h>

volatile int rd = 1;

const char *salt = NULL;
struct crypt_data input, output;

int verif_pwd(const char *pwd, const char *cipher_pwd, int safe)
{
    int r = 0;
    char *decrypted_pwd = NULL;

    switch(safe)
    {
      case 1:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      case 2:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      default:
        decrypted_pwd = crypt(pwd, cipher_pwd);
        break;
    }

    r = (strcmp(cipher_pwd, decrypted_pwd) == 0);

    return r;
}
```

In this example, `crypt_r` and `crypt` decrypt a password. However, `crypt` is nonreentrant and uses the unsafe Data Encryption Standard algorithm.

Correction — Use `crypt_r`

One possible correction is to replace `crypt` with `crypt_r`.

```
#define _GNU_SOURCE
#include <pwd.h>
#include <string.h>
#include <crypt.h>
```

```
volatile int rd = 1;

const char *salt = NULL;
struct crypt_data input, output;

int verif_pwd(const char *pwd, const char *cipher_pwd, int safe)
{
    int r = 0;
    char *decrypted_pwd = NULL;

    switch(safe)
    {
      case 1:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      case 2:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      default:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;
    }

    r = (strcmp(cipher_pwd, decrypted_pwd) == 0);

    return r;
}
```

Check Information

Group: Rec. 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MSC18-C

Introduced in R2019a

CERT C: Rec. MSC20-C

Do not use a switch statement to transfer control into a complex block

Description

Rule Definition

Do not use a switch statement to transfer control into a complex block.

Polyspace Implementation

This checker checks for **Switch label not at outermost level of body of switch statement**.

Examples

Switch label not at outermost level of body of switch statement

Issue

The issue occurs when you use a switch label and the most closely-enclosing compound statement is not the body of the switch statement. For instance a `case` label is enclosed inside a `for` loop that is enclosed inside the switch statement.

Risk

The C Standard permits placing a switch label (for instance, `case` or `default`) before any statement contained in the body of a switch statement. This flexibility can lead to unstructured code. To prevent unstructured code, make sure a switch label appears only at the outermost level of the body of a switch statement.

Check Information

Group: Rec. 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics
“Check for Coding Standard Violations”

External Websites
MSC20-C

Introduced in R2019a

CERT C: Rec. MSC21-C

Use robust loop termination conditions

Description

Rule Definition

Use robust loop termination conditions.

Polyspace Implementation

This checker checks for **Loop bounded with tainted value**.

Examples

Loop bounded with tainted value

Issue

Loop bounded with tainted value detects loops that are bounded by values from an unsecure source.

Risk

A tainted value can cause over looping or infinite loops. Attackers can use this vulnerability to crash your program or cause other unintended behavior.

Fix

Before starting the loop, validate unknown boundary and iterator values.

Example - Loop Boundary From Input Argument

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedloopboundary(int count) {
    int res = 0;
    for (int i=0 ; i < count; ++i) {
        res += i;
    }
    return res;
}
```

In this example, the function uses the input argument to loop `count` times. `count` could be any number because the value is not checked before starting the for-loop.

Correction — Check Loop Control

One possible correction is to check the value of the variable controlling the loop before starting the for-loop. This example checks if `count` is greater than zero and less than the maximum size.

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedloopboundary(int count) {
    int res = 0;

    if (count>0 && count<SIZE128) {
        for (int i=0 ; i<count ; ++i) {
            res += i;
        }
    }
    return res;
}
```

Check Information

Group: Rec. 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MSC21-C

Introduced in R2019a

CERT C: Rec. MSC22-C

Use the setjmp(), longjmp() facility securely

Description

Rule Definition

Use the setjmp(), longjmp() facility securely.

Polyspace Implementation

This checker checks for **Use of setjmp/longjmp**.

Examples

Use of setjmp/longjmp

Issue

Use of setjmp/longjmp occurs when you use a combination of `setjmp` and `longjmp` or `sigsetjmp` and `siglongjmp` to deviate from normal control flow and perform non-local jumps in your code.

Risk

Using `setjmp` and `longjmp`, or `sigsetjmp` and `siglongjmp` has the following risks:

- Nonlocal jumps are vulnerable to attacks that exploit common errors such as buffer overflows. Attackers can redirect the control flow and potentially execute arbitrary code.
- Resources such as dynamically allocated memory and open files might not be closed, causing resource leaks.
- If you use `setjmp` and `longjmp` in combination with a signal handler, unexpected control flow can occur. POSIX does not specify whether `setjmp` saves the signal mask.
- Using `setjmp` and `longjmp` or `sigsetjmp` and `siglongjmp` makes your program difficult to understand and maintain.

Fix

Perform nonlocal jumps in your code using `setjmp/longjmp` or `sigsetjmp/siglongjmp` only in contexts where such jumps can be performed securely. Alternatively, use POSIX threads if possible.

In C++, to simulate throwing and catching exceptions, use standard idioms such as `throw` expressions and `catch` statements.

Example - Use of `setjmp` and `longjmp`

```
#include <setjmp.h>
#include <signal.h>

extern int update(int);
extern void print_int(int);

static jmp_buf env;
```

```
void sighandler(int signum) {
    longjmp(env, signum);
}
void func_main(int i) {
    signal(SIGINT, sighandler);
    if (setjmp(env)==0) {
        while(1) {
            /* Main loop of program, iterates until SIGINT signal catch */
            i = update(i);
        }
    } else {
        /* Managing longjmp return */
        i = -update(i);
    }

    print_int(i);
    return;
}
```

In this example, the initial return value of `setjmp` is 0. The `update` function is called in an infinite `while` loop until the user interrupts it through a signal.

In the signal handling function, the `longjmp` statement causes a jump back to `main` and the return value of `setjmp` is now 1. Therefore, the `else` branch is executed.

Correction — Use Alternative to `setjmp` and `longjmp`

To emulate the same behavior more securely, use a `volatile` global variable instead of a combination of `setjmp` and `longjmp`.

```
#include <setjmp.h>
#include <signal.h>

extern int update(int);
extern void print_int(int);

volatile sig_atomic_t eflag = 0;

void sighandler(int signum) {
     eflag = signum;                   /* Fix: using global variable */
}

void func_main(int i) {
      /* Fix: Better design to avoid use of setjmp/longjmp */
    signal(SIGINT, sighandler);
    while(!eflag) {                    /* Fix: using global variable */
        /* Main loop of program, iterates until eflag is changed */
        i = update(i);
    }

    print_int(i);
    return;
}
```

Check Information

Group: Rec. 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MSC22-C

Introduced in R2019a

CERT C: Rec. MSC24-C

Do not use deprecated or obsolescent functions

Description

Rule Definition

Do not use deprecated or obsolescent functions.

Polyspace Implementation

This checker checks for **Use of obsolete standard function**.

Examples

Use of obsolete standard function

Issue

Use of obsolete standard function detects calls to standard function routines that are considered legacy, removed, deprecated, or obsolete by C/C++ coding standards.

Obsolete Function	Standards	Risk	Replacement Function
`asctime`	Deprecated in POSIX.1-2008	Not thread-safe.	`strftime` or `asctime_s`
`asctime_r`	Deprecated in POSIX.1-2008	Implementation based on unsafe function `sprintf`.	`strftime` or `asctime_s`
`bcmp`	Deprecated in 4.3BSD Marked as legacy in POSIX.1-2001.	Returns from function after finding the first differing byte, making it vulnerable to timing attacks.	`memcmp`
`bcopy`	Deprecated in 4.3BSD Marked as legacy in POSIX.1-2001.	Returns from function after finding the first differing byte, making it vulnerable to timing attacks.	`memcpy` or `memmove`
`brk` and `sbrk`	Marked as legacy in SUSv2 and POSIX.1-2001.		`malloc`
`bsd_signal`	Removed in POSIX.1-2008		`sigaction`
`bzero`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`memset`
`ctime`	Deprecated in POSIX.1-2008	Not thread-safe.	`strftime` or `asctime_s`

Obsolete Function	Standards	Risk	Replacement Function
`ctime_r`	Deprecated in POSIX.1-2008	Implementation based on unsafe function `sprintf`.	`strftime` or `asctime_s`
`cuserid`	Removed in POSIX.1-2001.	Not reentrant. Precise functionality not standardized causing portability issues.	`getpwuid`
`ecvt` and `fcvt`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008	Not reentrant	`snprintf`
`ecvt_r` and `fcvt_r`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008		`snprintf`
`ftime`	Removed in POSIX.1-2008		`time`, `gettimeofday`, `clock_gettime`
`gamma`, `gammaf`, `gammal`	Function not specified in any standard because of historical variations	Portability issues.	`tgamma`, `lgamma`
`gcvt`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`snprintf`
`getcontext`	Removed in POSIX.1-2008.	Portability issues.	Use POSIX thread instead.
`getdtablesize`	BSD API function not included in POSIX.1-2001	Portability issues.	`sysconf( _SC_OPEN_MAX )`
`gethostbyaddr`	Removed in POSIX.1-2008	Not reentrant	`getaddrinfo`
`gethostbyname`	Removed in POSIX.1-2008	Not reentrant	`getnameinfo`
`getpagesize`	BSD API function not included in POSIX.1-2001	Portability issues.	`sysconf( _SC_PAGESIZE )`
`getpass`	Removed in POSIX.1-2001.	Not reentrant.	`getpwuid`
`getw`	Not present in POSIX.1-2001.		`fread`
`getwd`	Marked legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`getcwd`
`index`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`strchr`
`makecontext`	Removed in POSIX.1-2008.	Portability issues.	Use POSIX thread instead.
`memalign`	Appears in SunOS 4.1.3. Not in 4.4 BSD or POSIX.1-2001		`posix_memalign`
`mktemp`	Removed in POSIX.1-2008.	Generated names are predictable and can cause a race condition.	`mkstemp` removes race risk

Obsolete Function	Standards	Risk	Replacement Function
`pthread_attr_getstackaddr` and `pthread_attr_setstackaddr`		Ambiguities in the specification of the `stackaddr` attribute cause portability issues	`pthread_attr_getstack` and `pthread_attr_setstack`
`putw`	Not present in POSIX.1-2001.	Portability issues.	`fwrite`
`qecvt` and `qfcvt`	Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008		`snprintf`
`qecvt_r` and `qfcvt_r`	Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008		`snprintf`
`rand_r`	Marked as obsolete in POSIX.1-2008		
`re_comp`	BSD API function	Portability issues	`regcomp`
`re_exes`	BSD API function	Portability issues	`regexec`
`rindex`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`strrchr`
`scalb`	Removed in POSIX.1-2008		`scalbln`, `scalblnf`, or `scalblnl`
`sigblock`	4.3BSD signal API whose origin is unclear		`sigprocmask`
`sigmask`	4.3BSD signal API whose origin is unclear		`sigprocmask`
`sigsetmask`	4.3BSD signal API whose origin is unclear		`sigprocmask`
`sigstack`	Interface is obsolete and not implemented on most platforms.	Portability issues.	`sigaltstack`
`sigvec`	4.3BSD signal API whose origin is unclear		`sigaction`
`swapcontext`	Removed in POSIX.1-2008	Portability issues.	Use POSIX threads.
`tmpnam` and `tmpnam_r`	Marked as obsolete in POSIX.1-2008.	This function generates a different string each time it is called, up to TMP_MAX times. If it is called more than TMP_MAX times, the behavior is implementation-defined.	`mkstemp`, `tmpfile`
`ttyslot`	Removed in POSIX.1-2001.		
`ualarm`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.	Errors are under-specified	`setitimer` or POSIX `timer_create`
`usleep`	Removed in POSIX.1-2008.		`nanosleep`
`utime`	SVr4, POSIX.1-2001. POSIX.1-2008 marks as obsolete.		

Obsolete Function	Standards	Risk	Replacement Function
`valloc`	Marked as obsolete in 4.3BSD. Marked as legacy in SUSv2. Removed from POSIX.1-2001		`posix_memalign`
`vfork`	Removed from POSIX.1-2008	Under-specified in previous standards.	`fork`
`wcswcs`	This function was not included in the final ISO/IEC 9899:1990/ Amendment 1:1995 (E).		`wcsstr`
`WinExec`	WinAPI provides this function only for 16-bit Windows compatibility.		`CreateProcess`
`LoadModule`	WinAPI provides this function only for 16-bit Windows compatibility.		`CreateProcess`

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Printing Out Time

```
#include <stdio.h>
#include <time.h>

void timecheck_bad(int argc, char *argv[])
{
    time_t ticks;

    ticks = time(NULL);
    printf("%.24s\r\n", ctime(&ticks));
}
```

In this example, the function `ctime` formats the current time and prints it out. However, `ctime` was removed after C99 because it does not work on multithreaded programs.

Correction — Different Time Function

One possible correction is to use `strftime` instead because this function uses a set buffer size.

```
#include <stdio.h>
#include <string.h>
#include <time.h>

void timecheck_good(int argc, char *argv[])
{
```

```
    char outBuff[1025];
    time_t ticks;
    struct tm * timeinfo;

    memset(outBuff, 0, sizeof(outBuff));

    ticks = time(NULL);
    timeinfo = localtime(&ticks);
    strftime(outBuff,sizeof(outBuff),"%I:%M%p.",timeinfo);
    fprintf(stdout, outBuff);
}
```

Check Information

Group: Rec. 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

MSC24-C

Introduced in R2019a

CERT C: Rec. POS05-C

Limit access to files by creating a jail

Description

Rule Definition

Limit access to files by creating a jail.

Polyspace Implementation

This checker checks for **File manipulation after chroot without chdir**.

Examples

File manipulation after chroot without chdir

Issue

File manipulation after `chroot()` without `chdir("/")` detects access to the file system outside of the jail created by `chroot`. By calling `chroot`, you create a file system jail that confines access to a specific file subsystem. However, this jail is ineffective if you do not call `chdir("/")`.

Risk

If you do not call `chdir("/")` after creating a `chroot` jail, file manipulation functions that takes a path as an argument can access files outside of the jail. An attacker can still manipulate files outside the subsystem that you specified, making the chroot jail ineffective.

Fix

After calling `chroot`, call `chdir("/")` to make your `chroot` jail more secure.

Example - Open File in `chroot`-jail

```
#include <unistd.h>
#include <stdio.h>

const char root_path[] = "/var/ftproot";
const char log_path[] = "file.log";
FILE* chrootmisuse() {
    FILE* res;
    chroot(root_path);
    chdir("base");
    res = fopen(log_path, "r");
    return res;
}
```

This example uses `chroot` to create a chroot-jail. However, to use the `chroot` jail securely, you must call `chdir("\")` afterward. This example calls `chdir("base")`, which is not equivalent. Bug Finder also flags `fopen` because `fopen` opens a file in the vulnerable `chroot`-jail.

Correction — Call `chdir("/")`

Before opening files, call `chdir("/")`.

```
#include <unistd.h>
#include <stdio.h>

const char root_path[] = "/var/ftproot";
const char log_path[] = "file.log";
FILE* chrootmisuse() {
    FILE* res;
    chroot(root_path);
    chdir("/");
    res = fopen(log_path, "r");
    return res;
}
```

Check Information

Group: Rec. 50. POSIX (POS)

See Also

`Check SEI CERT-C (-cert-c)`

Topics

“Check for Coding Standard Violations”

External Websites

POS05-C

Introduced in R2019a

CERT C: Rec. WIN00-C

Be specific when dynamically loading libraries

Description

Rule Definition

Be specific when dynamically loading libraries.

Polyspace Implementation

This checker checks for these issues:

- **Load of library from a relative path can be controlled by an external actor**.
- **Library loaded from externally controlled path**.

Examples

Load of library from a relative path can be controlled by an external actor

Issue

Load of library from a relative path can be controlled by an external actor detects library loading routines that load an external library. If you load the library using a relative path or no path, Bug Finder flags the loading routine as a defect.

Risk

By using a relative path or no path to load an external library, your program uses an unsafe search process to find the library. An attacker can control the search process and replace the intended library with a library of their own.

Fix

When you load an external library, specify the full path.

Example - Open Library with Library Name

```
#include <dlfcn.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <stdio.h>

void relative_path()
{
    dlopen("liberty.dll",RTLD_LAZY);
}
```

In this example, `dlopen` opens the `liberty` library by calling only the name of the library. However, this call to the library uses a relative path to find the library, which is unsafe.

Correction — Use Full Path to Library

One possible correction is to use the full path to the library when you load it into your program.

```
#include <dlfcn.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <stdio.h>

void relative_path()
{
    dlopen("/home/my_libs/library/liberty.dll",RTLD_LAZY);
}
```

Library loaded from externally controlled path

Issue

Library loaded from externally controlled path looks for libraries loaded from fixed or controlled paths. If unintended actors can control one or more locations on this fixed path, Bug Finder raises a defect.

Risk

If an attacker knows or controls the path that you use to load a library, the attacker can change:

- The library that the program loads, replacing the intended library and commands.
- The environment in which the library executes, giving unintended permissions and capabilities to the attacker.

Fix

When possible, use hard-coded or fully qualified path names to load libraries. It is possible the hard-coded paths do not work on other systems. Use a centralized location for hard-coded paths, so that you can easily modify the path within the source code.

Another solution is to use functions that require explicit paths. For example, `system()` does not require a full path because it can use the `PATH` environment variable. However, `execl()` and `execv()` do require the full path.

Example - Call Custom Library

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dlfcn.h>
#include <limits.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void* taintedpathlib() {
    void* libhandle = NULL;
```

```
    char lib[SIZE128] = "";
    char* userpath = getenv("LD_LIBRARY_PATH");
    strncpy(lib, userpath, SIZE128);
    strcat(lib, "/libX.so");
    libhandle = dlopen(lib, 0x00001);
    return libhandle;
}
```

This example loads the library `libX.so` from an environment variable `LD_LIBRARY_PATH`. An attacker can change the library path in this environment variable. The actual library you load could be a different library from the one that you intend.

Correction — Change and Check Path

One possible correction is to change how you get the library path and check the path of the library before opening the library. This example receives the path as an input argument. Then the path is checked to make sure the library is not under `/usr/`.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dlfcn.h>
#include <limits.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

/* Function to sanitize a string */
int sanitize_str(char* s, size_t n) {
    /* strlen is used here as a kind of firewall for tainted string errors */
    int res = (strlen(s) > 0 && strlen(s) < n);
    return res;
}
void* taintedpathlib(char* userpath) {
    void* libhandle = NULL;
    if (sanitize_str(userpath, SIZE128)) {
        char lib[SIZE128] = "";

        if (strncmp(userpath, "/usr", 4)!=0) {
            strncpy(lib, userpath, SIZE128);
            strcat(lib, "/libX.so");
            libhandle = dlopen(lib, RTLD_LAZY);
        }
    }
    return libhandle;
}
```

Check Information

Group: Rec. 51. Microsoft Windows (WIN)

See Also

`Check SEI CERT-C (-cert-c)`

Topics
“Check for Coding Standard Violations”

External Websites
WIN00-C

Introduced in R2019a

CERT C++ Rules

Acknowledgement

CERT C++: DCL30-C

Declare objects with appropriate storage durations

Description

Rule Definition

Declare objects with appropriate storage durations.

Polyspace Implementation

This checker checks for **Pointer or reference to stack variable leaving scope**.

Examples

Pointer or reference to stack variable leaving scope

Issue

Pointer or reference to stack variable leaving scope occurs when a pointer or reference to a local variable leaves the scope of the variable. For instance:

- A function returns a pointer to a local variable.
- A function performs the assignment `globPtr = &locVar`. `globPtr` is a global pointer variable and `locVar` is a local variable.
- A function performs the assignment `*paramPtr = &locVar`. `paramPtr` is a function parameter that is, for instance, an `int**` pointer and `locVar` is a local `int` variable.
- A C++ method performs the assignment `memPtr = &locVar`. `memPtr` is a pointer data member of the class the method belongs to. `locVar` is a variable local to the method.

The defect also applies to memory allocated using the `alloca` function. The defect does not apply to static, local variables.

Risk

Local variables are allocated an address on the stack. Once the scope of a local variable ends, this address is available for reuse. Using this address to access the local variable value outside the variable scope can cause unexpected behavior.

If a pointer to a local variable leaves the scope of the variable, Polyspace Bug Finder highlights the defect. The defect appears even if you do not use the address stored in the pointer. For maintainable code, it is a good practice to not allow the pointer to leave the variable scope. Even if you do not use the address in the pointer now, someone else using your function can use the address, causing undefined behavior.

Fix

Do not allow a pointer or reference to a local variable to leave the variable scope.

Example - Pointer to Local Variable Returned from Function

```
void func2(int *ptr) {
    *ptr = 0;
```

```
}

int* func1(void) {
    int ret = 0;
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

In this example, `func1` returns a pointer to local variable `ret`.

In `main`, `ptr` points to the address of the local variable. When `ptr` is accessed in `func2`, the access is illegal because the scope of `ret` is limited to `func1`,

Example - Pointer to Local Variable Escapes Through Lambda Expression

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [&] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

In this example, the `createAdder` function defines a lambda expression `adder` that captures the local variable `addThis` by reference. The scope of `addThis` is limited to the `createAdder` function. When the object returned by `createAdder` is called, a reference to the variable `addThis` is accessed outside its scope. When accessed in this way, the value of `addThis` is undefined.

Correction – Capture Local Variables by Copy in Lambda Expression Instead of Reference

If a function returns a lambda expression object, avoid capturing local variables by reference in the lambda object. Capture the variables by copy instead.

Variables captured by copy have the same lifetime as the lambda object, but variables captured by reference often have a smaller lifetime than the lambda object itself. When the lambda object is used, these variables accessed outside scope have undefined values.

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [=] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

Check Information

Group: 01. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

DCL30-C

Introduced in R2019a

CERT C++: DCL39-C

Avoid information leakage in structure padding

Description

Rule Definition

Avoid information leakage in structure padding.

Polyspace Implementation

This checker checks for **Information leak via structure padding**.

Examples

Information leak via structure padding

Issue

Information leak via structure padding occurs when you do not initialize the padding data of a structure or union before passing it across a trust boundary. A compiler adds padding bytes to the structure or union to ensure a proper memory alignment of its members. The bit-fields of the storage units can also have padding bits.

Information leak via structure padding raises a defect when:

- You call an untrusted function with structure or union pointer type argument containing uninitialized padding data.

  All external functions are considered untrusted.
- You copy or assign a structure or union containing uninitialized padding data to an untrusted object.

  All external structure or union objects, the output parameters of all externally linked functions, and the return pointer of all external functions are considered untrusted objects.

Risk

The padding bytes of the passed structure or union might contain sensitive information that an untrusted source can access.

Fix

- Prevent the addition of padding bytes for memory alignment by using the `pack` pragma or attribute supported by your compiler.
- Explicitly declare and initialize padding bytes as fields within the structure or union.
- Explicitly declare and initialize bit-fields corresponding to padding bits, even if you use the `pack` pragma or attribute supported by your compiler.

Example - Structure with Padding Bytes Passed to External Function

```
#include <stddef.h>
#include <stdlib.h>
```

```
#include <string.h>

typedef struct s_padding
{
 /* Padding bytes may be introduced between
 * 'char c' and 'int i'
 */
        char c;
    int i;

/*Padding bits may be introduced around the bit-fields
* even if you use "#pragma pack" (Windows) or
* __attribute__((__packed__)) (GNU)*/

    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;



/* External function */
extern void copy_object(void *out, void *in, size_t s);

void func(void *out_buffer)
{
/*Padding bytes not initialized*/

    S_Padding s = {'A', 10, 1, 3, {}};
/*Structure passed to external function*/

    copy_object((void *)out_buffer, (void *)&s, sizeof(s));
}

void main(void)
{
    S_Padding s1;
    func(&s1);
}
```

In this example, structure `s1` can have padding bytes between the `char c` and `int i` members. The bit-fields of the storage units of the structure can also contain padding bits. The content of the padding bytes and bits is accessible to an untrusted source when `s1` is passed to `func`.

Correction — Use pack Pragma to Prevent Padding Bytes

One possible correction in Microsoft Visual Studiois to use `#pragma pack()` to prevent padding bytes between the structure members. To prevent padding bits in the bit-fields of `s1`, explicitly declare and initialize the bit-fields even if you use `#pragma pack()`.

```
 #include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#define CHAR_BIT 8

#pragma pack(push, 1)
```

```
typedef struct s_padding
{
/*No Padding bytes when you use "#pragma pack" (Windows) or
* __attribute__((__packed__)) (GNU)*/
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
/* Padding bits explicitely declared */
    unsigned int bf_filler : sizeof(unsigned) * CHAR_BIT - 3;
    unsigned char buffer[20];
}

    S_Padding;

#pragma pack(pop)


/* External function */
extern void copy_object(void *out, void *in, size_t s);



void func(void *out_buffer)
{
    S_Padding s = {'A', 10, 1, 3, 0 /* padding bits */, {}};
    copy_object((void *)out_buffer, (void *)&s, sizeof(s));
}

void main(void)
{
    S_Padding s1;
    func(&s1);
}
```

Check Information

Group: 01. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

DCL39-C

Introduced in R2019a

CERT C++: DCL40-C

Do not create incompatible declarations of the same function or object

Description

Rule Definition

Do not create incompatible declarations of the same function or object.

Polyspace Implementation

This checker checks for **Declaration mismatch**.

Examples

Declaration mismatch

Issue

Declaration mismatch occurs when a function or variable declaration does not match other instances of the function or variable.

Risk

When a mismatch occurs between two variable declarations in different compilation units, a typical linker follows an algorithm to pick one declaration for the variable. If you expect a variable declaration that is different from the one chosen by the linker, you can see unexpected results when the variable is used.

A similar issue can occur with mismatch in function declarations.

Fix

The fix depends on the type of declaration mismatch. If both declarations indeed refer to the same object, use the same declaration. If the declarations refer to different objects, change the names of the one of the variables. If you change a variable name, remember to make the change in all places that use the variable.

Sometimes, declaration mismatches can occur because the declarations are affected by previous preprocessing directives. For instance, a declaration occurs in a macro, and the macro is defined on one inclusion path but undefined in another. These declaration mismatches can be tricky to debug. Identify the divergence between the two inclusion paths and fix the conflicting macro definitions.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Inconsistent Declarations in Two Files

file1.c

```
int foo(void) {
    return 1;
}
```

file2.c

```
double foo(void);

int bar(void) {
    return (int)foo();
}
```

In this example, *file1.c* declares `foo()` as returning an integer. In *file2.c*, `foo()` is declared as returning a double. This difference raises a defect on the second instance of `foo` in *file2*.

Correction — Align the Function Return Values

One possible correction is to change the function declarations so that they match. In this example, by changing the declaration of `foo` in *file2.c* to match *file1.c*, the defect is fixed.

file1.c

```
int foo(void) {
    return 1;
}
```

file2.c

```
int foo(void);

int bar(void) {
    return foo();
}
```

Example - Inconsistent Structure Alignment

test1.c

```
#include "square.h"
#include "circle.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

test2.c

```
#include "circle.h"
#include "square.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

circle.h

```
#pragma pack(1)

extern struct aCircle{
    int radius;
} circle;
```

square.h

```
extern struct aSquare {
    unsigned int side:1;
} square;
```

In this example, a declaration mismatch defect is raised on `square` in *square.h* because Polyspace infers that `square` in *square.h* does not have the same alignment as `square` in *test2.c*. This error occurs because the `#pragma pack(1)` statement in *circle.h* declares specific alignment. In *test2.c*, *circle.h* is included before *square.h*. Therefore, the `#pragma pack(1)` statement from *circle.h* is not reset to the default alignment after the `aCircle` structure. Because of this omission, *test2.c* infers that the `aSquare square` structure also has an alignment of `1` byte.

Correction — Close Packing Statements

One possible correction is to reset the structure alignment after the `aCircle` struct declaration. For the GNU or Microsoft Visual compilers, fix the defect by adding a `#pragma pack()` statement at the end of *circle.h*.

test1.c

```
#include "square.h"
#include "circle.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

test2.c

```
#include "circle.h"
#include "square.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

circle.h

```
#pragma pack(1)

extern struct aCircle{
    int radius;
} circle;

#pragma pack()
```

square.h

```
extern struct aSquare {
    unsigned int side:1;
} square;
```

Other compilers require different `#pragma pack` syntax. For your syntax, see the documentation for your compiler.

Correction — Use the `Ignore pragma pack directives` Option

One possible correction is to add the `Ignore pragma pack directives` option to your Bug Finder analysis. If you want the structure alignment to change for each structure, and you do not want to see this **Declaration mismatch** defect, use this correction.

1. On the Configuration pane, select the **Advanced Settings** pane.
2. In the **Other** box, enter `-ignore-pragma-pack`.
3. Rerun your analysis.

   The **Declaration mismatch** defect is resolved.

Check Information

Group: 01. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

DCL40-C

Introduced in R2019a

CERT C++: DCL50-CPP

Do not define a C-style variadic function

Description

Rule Definition

Do not define a C-style variadic function.

Polyspace Implementation

This checker checks for **Function definition with ellipsis notation**.

Examples

Function definition with ellipsis notation

Issue

The issue occurs when you define a function using the ellipsis notation.

```
int func( cont char* format, ...);
```

Check Information

Group: 01. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics
“Check for Coding Standard Violations”

External Websites
DCL50-CPP

Introduced in R2019a

CERT C++: DCL51-CPP

Do not declare or define a reserved identifier

Description

Rule Definition

Do not declare or define a reserved identifier.

Polyspace Implementation

This checker checks for **Defining reserved identifier**.

Examples

Defining reserved identifier

Issue

The issue occurs when you define, redefine, or undefine a reserved identifier, macro, or function in the standard library.

Check Information

Group: 01. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics
“Check for Coding Standard Violations”

External Websites
DCL51-CPP

Introduced in R2019a

CERT C++: DCL52-CPP

Never qualify a reference type with const or volatile

Description

Rule Definition

Never qualify a reference type with const or volatile.

Polyspace Implementation

This checker checks for these issues:

- **const-Qualified Reference Type**.
- **Modification of const-qualified Reference Types**.

Examples

`const`-Qualified Reference Type

```
int func (int &const iRef) {
    iRef++;
    return iRef%2;
}
```

In this example, `iRef` is a `const`-qualified reference type. Since `iRef` cannot refer to another variable, the `const` qualifier is redundant.

Issue

Remove the redundant `const` qualifier. Since `iRef` is modified in `func`, it is not meant to refer to a `const`-qualified variable. Moving the `const` qualifier before & will cause a compilation error.

```
int func (int &iRef) {
    iRef++;
    return iRef%2;
}
```

Correction — Fix Placement of `const` Qualifier

If you do not identify to modify `iRef` in `func`, declare `iRef` as a reference to a `const`-qualified variable. Place the `const` qualifier before the & operator. Make sure you do not modify `iRef` in `func`.

```
int func (int const &iRef) {
    return (iRef+1)%2;
}
```

Modification of `const`-qualified Reference Types

```
typedef const int cint;
typedef cint& ref_to_cint;

void func(ref_to_cint refVal, int initVal){
```

```
   refVal = val;
}
```

In this example, `ref_to_cint` is a reference to a `const`-qualified type. The variable `refVal` of type `ref_to_cint` is supposed to be initialized when `func` is called and not modified subsequently. The modification violates the contract implied by the `const` qualifier.

Issue

One possible correction is to avoid the `const` in the declaration of the reference type.

```
typedef int& ref_to_int;

void func(ref_to_int refVal, int initVal){
   refVal = val;
}
```

Check Information

Group: 01. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

DCL52-CPP

Introduced in R2019a

CERT C++: DCL53-CPP

Do not write syntactically ambiguous declarations

Description

Rule Definition

Do not write syntactically ambiguous declarations.

Polyspace Implementation

This checker checks for these issues:

- **Function or Object Declaration**.
- **Unnamed Object or Unnamed Function Parameter Declaration**.
- **Unnamed Object or Named Function Parameter Declaration**.

Examples

Function or Object Declaration

```
class ResourceType {
      int aMember;
    public:
      int getMember();
};

void getResource() {
    ResourceType aResource();
}
```

In this example, `aResource` might be used as an object but the declaration syntax indicates a function declaration.

Issue

One possible correction (after C++11) is to use braces for object declaration.

```
class ResourceType {
      int aMember;
    public:
      int getMember();
};

void getResource() {
    ResourceType aResource{};
}
```

Unnamed Object or Unnamed Function Parameter Declaration

```
class MemberType {};

class ResourceType {
```

```
      MemberType aMember;
    public:
      ResourceType(MemberType m) {aMember = m;}
      int getMember();
};

void getResource() {
    ResourceType aResource(MemberType());
}
```

In this example, `aResource` might be used as an object initialized with an unnamed object of type `MemberType` but the declaration syntax indicates a function with an unnamed parameter of function pointer type. The function pointer points to a function with no arguments and type `MemberType`.

Issue

One possible correction (after C++11) is to use braces for object declaration.

```
class MemberType {};

class ResourceType {
      MemberType aMember;
    public:
      ResourceType(MemberType m) {aMember = m;}
      int getMember();
};

void getResource {
    ResourceType aResource{MemberType()};
}
```

Unnamed Object or Named Function Parameter Declaration

```
class Integer {
       int aMember;
    public:
       Integer(int d) {aMember = d;}
       int getMember();
};

int aInt = 0;
Integer aInteger(Integer(aInt));
```

In this example, `aInteger` might be an object constructed with an unnamed object `Integer(aInt)` (an object of class `Integer` which itself is constructed using the variable `aInt`). However, the declaration syntax indicates that `aInteger` is a function with a named parameter `aInt` of type `Integer` (the superfluous parenthesis is ignored).

Issue

One possible correction (after C++11) is to use {} for object declaration.

```
class Integer {
       int aMember;
    public:
       Integer(int d) {aMember = d;}
       int getMember();
};
```

```
int aInt = 0;
Integer aInteger{Integer{aInt}};
```

Correction — Remove Superfluous Parenthesis for Named Parameter Declaration

If `aInteger` is a function with a named parameter `aInt`, remove the superfluous () around `aInt`.

```
class Integer {
       int aMember;
    public:
       Integer(int d) {aMember = d;}
       int getMember();
};

Integer aInteger(Integer aInt);
```

Check Information

Group: 01. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

DCL53-CPP

Introduced in R2019a

CERT C++: DCL54-CPP

Overload allocation and deallocation functions as a pair in the same scope

Description

Rule Definition

Overload allocation and deallocation functions as a pair in the same scope.

Polyspace Implementation

This checker checks for **Mismatch Between Overloaded operator new and operator delete**.

Examples

Mismatch Between Overloaded `operator new` and `operator delete`

```
#include <new>
#include <cstdlib>

int global_store;

void update_bookkeeping(void *allocated_ptr, bool alloc) {
   if(alloc)
      global_store++;
   else
      global_store--;
}


void *operator new(std::size_t size, const std::nothrow_t& tag);
void *operator new(std::size_t size, const std::nothrow_t& tag) {
    void *ptr = (void*)malloc(size);
    if (ptr != nullptr)
        update_bookkeeping(ptr, true);
    return ptr;
}

void operator delete[](void *ptr, const std::nothrow_t& tag);
void operator delete[](void* ptr, const std::nothrow_t& tag) {
    update_bookkeeping(ptr, false);
    free(ptr);
}
```

In this example, the operators `operator new` and `operator delete[]` are overloaded but there are no overloads of the corresponding `operator delete` and `operator new[]` operators.

The overload of `operator new` calls a function `update_bookkeeping` to change the value of a global variable `global_store`. If the default `operator delete` is called, this global variable is unaffected, which might defy developer's expectations.

Issue

If you want to overload `operator new`, overload the corresponding form of `operator delete` in the same scope.

```
#include <new>
#include <cstdlib>

int global_store;

void update_bookkeeping(void *allocated_ptr, bool alloc) {
   if(alloc)
      global_store++;
   else
      global_store--;
}


void *operator new(std::size_t size, const std::nothrow_t& tag);
void *operator new(std::size_t size, const std::nothrow_t& tag) {
    void *ptr = (void*)malloc(size);
    if (ptr != nullptr)
        update_bookkeeping(ptr, true);
    return ptr;
}

void operator delete(void *ptr, const std::nothrow_t& tag);
void operator delete(void* ptr, const std::nothrow_t& tag) {
    update_bookkeeping(ptr, false);
    free(ptr);
}
```

Check Information

Group: 01. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

DCL54-CPP

Introduced in R2019a

CERT C++: DCL57-CPP

Do not let exceptions escape from destructors or deallocation functions

Description

Rule Definition

Do not let exceptions escape from destructors or deallocation functions.

Polyspace Implementation

This checker checks for **Class destructor exiting with an exception**.

Examples

Class destructor exiting with an exception

Issue

The checker flags exceptions thrown in the body of the destructor. If the destructor calls another function, the checker does not detect if that function throws an exception.

The checker does not detect these situations:

- A `catch` statement does not catch exceptions of all types that are thrown.

  The checker considers the presence of a `catch` statement corresponding to a `try` block as indication that an exception is caught.
- `throw` statements inside `catch` blocks

Check Information

Group: 01. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

DCL57-CPP

Introduced in R2019a

CERT C++: DCL58-CPP

Do not modify the standard namespaces

Description

Rule Definition

Do not modify the standard namespaces.

Polyspace Implementation

This checker checks for **Modification of standard namespaces**.

Examples

Modification of standard namespaces

Issue

Modification of standard namespaces occurs when you make additions to the namespaces `std`, `posix`, or their subspaces, or you specialize class or function templates from these namespaces.

Risk

Adding declarations or definitions to namespace `std` or its subspaces, or to `posix` or its subspaces, leads to undefined behavior. Likewise, explicitly specializing a member function or member class of a standard library leads to undefined behavior.

The standard allows exceptions to the specialization aspect of the rule for standard library templates that require a user-defined type. If you have a process that all rule violations must be justified and an issue flagged by the checker belongs to this category of exceptions, justify the issue using comments in your result or code. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Check Information

Group: 01. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

DCL58-CPP

Introduced in R2019b

CERT C++: DCL59-CPP

Do not define an unnamed namespace in a header file

Description

Rule Definition

Do not define an unnamed namespace in a header file.

Polyspace Implementation

This checker checks for **Unnamed namespace in header file**.

Examples

Unnamed namespace in header file

Issue

Unnamed namespace in header file detects an unnamed namespace in a header file, which can lead to multiple definitions of objects in the namespace.

Risk

According to the C++ standard, names in an unnamed namespace, for instance, `aVar` here:

```
namespace {
   int aVar;
}
```

have internal linkage by default. If a header file contains an unnamed namespace, each translation unit `#include`-ing the header file defines its own instance of objects in the namespace. The multiple definitions are probably not what you intended and can lead to unexpected results, undesired memory usage or inadvertently violating the one-definition rule.

Fix

Specify names for namespaces in header files or avoid using namespaces in header files.

Example – Unexpected Results from Unnamed Namespaces in Header Files

Header File: `aHeader.h`

```
namespace {
   int aVar;
}
```

First source file: `aSource.cpp`

```
#include "aHeader.h"
#include <iostream>

void setVar(int arg) {
    std::cout << "Current value: " << aVar << std::endl;
```

```
    aVar = arg;
    std::cout << "Value set at: " << aVar << std::endl;
}
```

Second source file: `anotherSource.cpp`

```
#include "aHeader.h"
#include <iostream>

extern void setVar(int);

void resetVar() {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = 0;
    std::cout << "Value set at: 0" << std::endl;
}

void main() {
    setVar(1);
    resetVar();
}
```

In this example, the unnamed namespace leads to two definitions of `aVar` in the translation unit from `aSource.cpp` and the translation unit from `anotherSource.cpp`. The two definitions lead to the possibly unexpected output:

```
Current value: 0
Value set at: 1
Current value: 0
Value set at: 0
```

Correction – Avoid the Unnamed Namespace

One possible correction is to simply avoid a namespace in the header file.

Header File: `aHeader.h`

```
extern int aVar;
```

First source file: `aSource.cpp`

```
#include "aHeader.h"
#include <iostream>

void setVar(int arg) {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = arg;
    std::cout << "Value set at: " << aVar << std::endl;
}
```

Second source file: `anotherSource.cpp`

```
#include "aHeader.h"
#include <iostream>

extern void setVar(int);
int aVar;

void resetVar() {
```

```
    std::cout << "Current value: " << aVar << std::endl;
    aVar = 0;
    std::cout << "Value set at: 0" << std::endl;
}

void main() {
    setVar(1);
    resetVar();
}
```

You now see the expected sequence in the output:

```
Current value: 0
Value set at: 1
Current value: 1
Value set at: 0
```

Check Information

Group: 01. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

DCL59-CPP

Introduced in R2019a

CERT C++: DCL60-CPP

Obey the one-definition rule

Description

Rule Definition

Obey the one-definition rule.

Polyspace Implementation

This checker checks for **Inline constraint not respected**.

Examples

Inline constraint not respected

Issue

Inline constraint not respected occurs when you refer to a file scope modifiable static variable or define a local modifiable static variable in a nonstatic inlined function. The checker considers a variable as modifiable if it is not `const`-qualified.

For instance, `var` is a modifiable `static` variable defined in an `inline` function `func`. `g_step` is a file scope modifiable static variable referred to in the same inlined function.

```
static int g_step;
inline void func (void) {
   static int var = 0;
   var += g_step;
}
```

Risk

When you modify a static variable in multiple function calls, you expect to modify the same variable in each call. For instance, each time you call `func`, the same instance of `var1` is incremented but a separate instance of `var2` is incremented.

```
void func(void) {
   static var1 = 0;
   var2 = 0;
   var1++;
   var2++;
}
```

If a function has an inlined and non-inlined definition (in separate files), when you call the function, the C standard allows compilers to use either the inlined or the non-inlined form (see ISO/IEC 9899:2011, sec. 6.7.4). If your compiler uses an inlined definition in one call and the non-inlined definition in another, you are no longer modifying the same variable in both calls. This behavior defies the expectations from a static variable.

Fix

Use one of these fixes:

- If you do not intend to modify the variable, declare it as `const`.

  If you do not modify the variable, there is no question of unexpected modification.
- Make the variable non-`static`. Remove the `static` qualifier from the declaration.

  If the variable is defined in the function, it becomes a regular local variable. If defined at file scope, it becomes an extern variable. Make sure that this change in behavior is what you intend.
- Make the function `static`. Add a `static` qualifier to the function definition.

  If you make the function `static`, the file with the inlined definition always uses the inlined definition when the function is called. Other files use another definition of the function. The question of which function definition gets used is not left to the compiler.

Example - Static Variable Use in Inlined and External Definition

```
/* file1. c  : contains inline definition of get_random()*/

inline unsigned int get_random(void)
{

    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}


int call_get_random(void)
{
    unsigned int rand_no;
    int ii;
    for (ii = 0; ii < 100; ii++) {
         rand_no = get_random();
    }
    rand_no = get_random();
    return 0;
}

/* file2. c  : contains external definition of get_random()*/

extern unsigned int get_random(void)
{
    /* Initialize seeds */
    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}
```

In this example, `get_random()` has an inline definition in `file1.c` and an external definition in `file2.c`. When `get_random` is called in `file1.c`, compilers are free to choose whether to use the inline or the external definition.

Depending on the definition used, you might or might not modify the version of `m_z` and `m_w` in the inlined version of `get_random()`. This behavior contradicts the usual expectations from a static variable. When you call `get_random()`, you expect to always modify the same `m_z` and `m_w`.

Correction — Make Inlined Function Static

One possible correction is to make the inlined `get_random()` static. Irrespective of your compiler, calls to `get_random()` in `file1.c` then use the inlined definition. Calls to `get_random()` in other files use the external definition. This fix removes the ambiguity about which definition is used and whether the static variables in that definition are modified.

```
/* file1. c  : contains inline definition of get_random()*/

static inline unsigned int get_random(void)
{

    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}


int call_get_random(void)
{
    unsigned int rand_no;
    int ii;
    for (ii = 0; ii < 100; ii++) {
         rand_no = get_random();
    }
    rand_no = get_random();
    return 0;
}

/* file2. c  : contains external definition of get_random()*/

extern unsigned int get_random(void)
{
    /* Initialize seeds */
    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}
```

Check Information

Group: 01. Declarations and Initialization (DCL)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

DCL60-CPP

Introduced in R2019a

CERT C++: EXP34-C

Do not dereference null pointers

Description

Rule Definition

Do not dereference null pointers.

Polyspace Implementation

This checker checks for **Null pointer**.

Examples

Null pointer

Issue

Null pointer occurs when you use a pointer with a value of `NULL` as if it points to a valid memory location.

Risk

Dereferencing a null pointer is undefined behavior. In most implementations, the dereference can cause your program to crash.

Fix

Check a pointer for `NULL` before dereference.

If the issue occurs despite an earlier check for `NULL`, look for intermediate events between the check and the subsequent dereference. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

Example - Null pointer error

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 int* p=NULL;

 *p=arr[0];
 /* Defect: Null pointer dereference */

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
```

```
  }

 return *p;
}
```

The pointer `p` is initialized with value of `NULL`. However, when the value `arr[0]` is written to `*p`, `p` is assumed to point to a valid memory location.

Correction — Assign Address to Null Pointer Before Dereference

One possible correction is to initialize `p` with a valid memory address before dereference.

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 /* Fix: Assign address to null pointer */
 int* p=&arr[0];

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

EXP34-C

Introduced in R2019a

CERT C++: EXP35-C

Do not modify objects with temporary lifetime

Description

Rule Definition

Do not modify objects with temporary lifetime.

Polyspace Implementation

This checker checks for **Accessing object with temporary lifetime**.

Examples

Accessing object with temporary lifetime

Issue

Accessing object with temporary lifetime occurs when you attempt to read from or write to an object with temporary lifetime that is returned by a function call. In a structure or union returned by a function, and containing an array, the array members are temporary objects. The lifetime of temporary objects ends:

- When the full expression or full declarator containing the call ends, as defined in the C11 Standard.
- After the next sequence point, as defined in the C90 and C99 Standards. A sequence point is a point in the execution of a program where all previous evaluations are complete and no subsequent evaluation has started yet.

For C++ code, **Accessing object with temporary lifetime** raises a defect only when you write to an object with a temporary lifetime.

If the temporary lifetime object is returned by address, no defect is raised.

Risk

Modifying objects with temporary lifetime is undefined behavior and can cause abnormal program termination and portability issues.

Fix

Assign the object returned from the function call to a local variable. The content of the temporary lifetime object is copied to the variable. You can now modify it safely.

Example - Modifying Temporary Lifetime Object Returned by Function Call

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6
```

```
struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

/* func_temp() returns a struct value containing
* an array with a temporary lifetime.
*/
int func(void) {

/*Writing to temporary lifetime object is
 undefined behavior
 */
    return ++(func_temp().a[0]);
}

void main(void) {
    (void)func();
}
```

In this example, `func_temp()` returns by value a structure with an array member `a`. This member has temporary lifetime. Incrementing it is undefined behavior.

Correction — Assign Returned Value to Local Variable Before Writing

One possible correction is to assign the return of the call to `func_temp()` to a local variable. The content of the temporary object `a` is copied to the variable, which you can safely increment.

```
 #include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6

struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

int func(void) {

/* Assign object returned by function call to
 *local variable
 */
    struct S_Array s = func_temp();

/* Local variable can safely be
 *incremented
 */
    ++(s.a[0]);
```

```
    return s.a[0];
}

void main(void) {
    (void)func();
}
```

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

EXP35-C

Introduced in R2019a

CERT C++: EXP36-C

Do not cast pointers into more strictly aligned pointer types

Description

Rule Definition

Do not cast pointers into more strictly aligned pointer types.

Polyspace Implementation

This checker checks for **Wrong allocated object size for cast**.

Examples

Wrong allocated object size for cast

Issue

Wrong allocated object size for cast occurs during pointer conversion when the pointer's address is misaligned. If a pointer is converted to a different pointer type, the size of the allocated memory must be a multiple of the size of the destination pointer.

Risk

Dereferencing a misaligned pointer has undefined behavior and can cause your program to crash.

Fix

Suppose you convert a pointer `ptr1` to `ptr2`. If `ptr1` points to a buffer of `N` bytes and `ptr2` is a *`type`* * pointer where `sizeof(`*`type`*`)` is `n` bytes, make sure that `N` is an integer multiple of `n`.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Dynamic Allocation of Pointers

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(13);
    long *dest;

    dest = (long*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to a `long*`. The dynamically allocated memory of `ptr`, 13 bytes, is not a multiple of the size of `dest`, 4 bytes. This misalignment causes the **Wrong allocated object size for cast** defect.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the allocated memory to 12 instead of 13.

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(12);
    long *dest;

    dest = (long*)ptr;
}
```

Example - Static Allocation of Pointers

```
void static_non_align(void){
    char arr[13], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to an `int*` in line 6. `ptr` has a memory size of 13 bytes because the array `arr` has a size of 13 bytes. The size of `dest` is 4 bytes, which is not a multiple of 13. This misalignment causes the **Wrong allocated object size for cast** defect.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the size of the array `arr` to a multiple of 4.

```
void static_non_align(void){
    char arr[12], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr;
}
```

Example - Allocation with a Function

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(13);  //defect
    dest2 = (char*)my_alloc(13); //not a defect
}
```

In this example, the software raises a defect on the conversion of the pointer returned by `my_alloc(13)` to an `int*` in line 11. `my_alloc(13)` returns a pointer with a dynamically allocated size of 13 bytes. The size of `dest1` is 4 bytes, which is not a divisor of 13. This misalignment causes the **Wrong allocated object size for cast** defect. In line 12, the same function call, `my_alloc(13)`, does not call a defect for the conversion to `dest2` because the size of `char*`, 1 byte, a divisor of 13.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the argument for `my_alloc` to a multiple of 4.

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(12);
    dest2 = (char*)my_alloc(13);
}
```

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

EXP36-C

Introduced in R2019a

CERT C++: EXP37-C

Call functions with the correct number and type of arguments

Description

Rule Definition

Call functions with the correct number and type of arguments.

Polyspace Implementation

This checker checks for these issues:

- **Bad file access mode or status**.
- **Unreliable cast of function pointer**.
- **Standard function call with incorrect arguments**.

Examples

Bad file access mode or status

Issue

Bad file access mode or status occurs when you use functions in the `fopen` or `open` group with invalid or incompatible file access modes, file creation flags, or file status flags as arguments. For instance, for the `open` function, examples of valid:

- Access modes include `O_RDONLY`, `O_WRONLY`, and `O_RDWR`
- File creation flags include `O_CREAT`, `O_EXCL`, `O_NOCTTY`, and `O_TRUNC`.
- File status flags include `O_APPEND`, `O_ASYNC`, `O_CLOEXEC`, `O_DIRECT`, `O_DIRECTORY`, `O_LARGEFILE`, `O_NOATIME`, `O_NOFOLLOW`, `O_NONBLOCK`, `O_NDELAY`, `O_SHLOCK`, `O_EXLOCK`, `O_FSYNC`, `O_SYNC` and so on.

The defect can occur in the following situations.

Situation	Risk	Fix
You pass an empty or invalid access mode to the `fopen` function. According to the ANSI C standard, the valid access modes for `fopen` are: • `r`,`r+` • `w`,`w+` • `a`,`a+` • `rb`, `wb`, `ab` • `r+b`, `w+b`, `a+b` • `rb+`, `wb+`, `ab+`	`fopen` has undefined behavior for invalid access modes. Some implementations allow extension of the access mode such as: • GNU: `rb+cmxe,ccs=utf` • Visual C++: `a+t`, where `t` specifies a text mode. However, your access mode string must begin with one of the valid sequences.	Pass a valid access mode to `fopen`.
You pass the status flag `O_APPEND` to the `open` function without combining it with either `O_WRONLY` or `O_RDWR`.	`O_APPEND` indicates that you intend to add new content at the end of a file. However, without `O_WRONLY` or `O_RDWR`, you cannot write to the file. The `open` function does not return -1 for this logical error.	Pass either `O_APPEND|O_WRONLY` or `O_APPEND|O_RDWR` as access mode.
You pass the status flags `O_APPEND` and `O_TRUNC` together to the `open` function.	`O_APPEND` indicates that you intend to add new content at the end of a file. However, `O_TRUNC` indicates that you intend to truncate the file to zero. Therefore, the two modes cannot operate together. The `open` function does not return -1 for this logical error.	Depending on what you intend to do, pass one of the two modes.
You pass the status flag `O_ASYNC` to the `open` function.	On certain implementations, the mode `O_ASYNC` does not enable signal-driven I/O operations.	Use the `fcntl(pathname, F_SETFL, O_ASYNC);` instead.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Invalid Access Mode with `fopen`

```
#include <stdio.h>

void func(void) {
    FILE *file = fopen("data.txt", "rw");
    if(file!=NULL) {
        fputs("new data",file);
        fclose(file);
    }
}
```

In this example, the access mode `rw` is invalid. Because `r` indicates that you open the file for reading and `w` indicates that you create a new file for writing, the two access modes are incompatible.

Correction — Use Either `r` or `w` as Access Mode

One possible correction is to use the access mode corresponding to what you intend to do.

```
#include <stdio.h>

void func(void) {
    FILE *file = fopen("data.txt", "w");
    if(file!=NULL) {
        fputs("new data",file);
        fclose(file);
    }
}
```

Unreliable cast of function pointer

Issue

Unreliable cast of function pointer occurs when a function pointer is cast to another function pointer that has different argument or return type.

This defect applies only if the code language for the project is C.

Risk

If you cast a function pointer to another function pointer with different argument or return type and then use the latter function pointer to call a function, the behavior is undefined.

Fix

Avoid a cast between two function pointers with mismatch in argument or return types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Unreliable cast of function pointer error

```
#include <stdio.h>
#include <math.h>
#include <stdio.h>
#define PI 3.142
```

```
double Calculate_Sum(int (*fptr)(double))
{
    double  sum = 0.0;
    double  y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;

    fp = sin;
    sum = Calculate_Sum(fp);
    /* Defect: fp implicitly cast to int(*) (double) */

    printf("sum(sin): %f\n", sum);
    return 0;
}
```

The function pointer `fp` is declared as `double (*)(double)`. However in passing it to function `Calculate_Sum`, `fp` is implicitly cast to `int (*)(double)`.

Correction — Avoid Function Pointer Cast

One possible correction is to check that the function pointer in the definition of `Calculate_Sum` has the same argument and return type as `fp`. This step makes sure that `fp` is not implicitly cast to a different argument or return type.

```
#include <stdio.h>
#include <math.h>
#include <stdio.h>
# define PI 3.142

/*Fix: fptr has same argument and return type everywhere*/
double Calculate_Sum(double (*fptr)(double))
{
    double  sum = 0.0;
    double y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;
```

```
    fp = sin;
    sum = Calculate_Sum(fp);
    printf("sum(sin): %f\n", sum);

    return 0;
}
```

Standard function call with incorrect arguments

Issue

Standard function call with incorrect arguments occurs when the arguments to certain standard functions do not meet the requirements for their use in the functions.

For instance, the arguments to these functions can be invalid in the following ways.

Function Type	Situation	Risk	Fix
String manipulation functions such as `strlen` and `strcpy`	The pointer arguments do not point to a `NULL`-terminated string.	The behavior of the function is undefined.	Pass a `NULL`-terminated string to string manipulation functions.
File handling functions in `stdio.h` such as `fputc` and `fread`	The `FILE*` pointer argument can have the value `NULL`.	The behavior of the function is undefined.	Test the `FILE*` pointer for `NULL` before using it as function argument.
File handling functions in `unistd.h` such as `lseek` and `read`	The file descriptor argument can be -1.	The behavior of the function is undefined. Most implementations of the `open` function return a file descriptor value of -1. In addition, they set `errno` to indicate that an error has occurred when opening a file.	Test the return value of the `open` function for -1 before using it as argument for `read` or `lseek`. If the return value is -1, check the value of `errno` to see which error has occurred.
	The file descriptor argument represents a closed file descriptor.	The behavior of the function is undefined.	Close the file descriptor only after you have completely finished using it. Alternatively, reopen the file descriptor before using it as function argument.
Directory name generation functions such as `mkdtemp` and `mkstemps`	The last six characters of the string template are not `XXXXXX`.	The function replaces the last six characters with a string that makes the file name unique. If the last six characters are not `XXXXXX`, the function cannot generate a unique enough directory name.	Test if the last six characters of a string are `XXXXXX` before using the string as function argument.

Function Type	Situation	Risk	Fix
Functions related to environment variables such as `getenv` and `setenv`	The string argument is `""`.	The behavior is implementation-defined.	Test the string argument for `""` before using it as `getenv` or `setenv` argument.
	The string argument terminates with an equal sign, =. For instance, `"C="` instead of `"C"`.	The behavior is implementation-defined.	Do not terminate the string argument with =.
String handling functions such as `strtok` and `strstr`	• `strtok`: The delimiter argument is `""`. • `strstr`: The search string argument is `""`.	Some implementations do not handle these edge cases.	Test the string for `""` before using it as function argument.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - `NULL` Pointer Passed as `strnlen` Argument

```
#include <string.h>
#include <stdlib.h>

enum {
    SIZE10 = 10,
    SIZE20 = 20
};

int func() {
    char* s = NULL;
    return strnlen(s, SIZE20);
}
```

In this example, a `NULL` pointer is passed as `strnlen` argument instead of a `NULL`-terminated string.

Before running analysis on the code, specify a GNU compiler. See `Compiler (-compiler)`.

Correction — Pass `NULL`-terminated String

Pass a `NULL`-terminated string as the first argument of `strnlen`.

```
#include <string.h>
#include <stdlib.h>
```

```
enum {
    SIZE10 = 10,
    SIZE20 = 20
};

int func() {
    char* s = "";
    return strnlen(s, SIZE20);
}
```

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

EXP37-C

Introduced in R2019a

CERT C++: EXP39-C

Do not access a variable through a pointer of an incompatible type

Description

Rule Definition

Do not access a variable through a pointer of an incompatible type.

Polyspace Implementation

This checker checks for **Pointer conversion to unrelated pointer type**.

Examples

Pointer conversion to unrelated pointer type

Issue

The checker flags all pointer conversions including between a pointer to a `struct` object and a pointer to the first member of the same `struct` type.

Indirect conversions from a pointer to non-pointer type are not detected.

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics
“Check for Coding Standard Violations”

External Websites
EXP39-C

Introduced in R2019a

CERT C++: EXP42-C

Do not compare padding data

Description

Rule Definition

Do not compare padding data.

Polyspace Implementation

This checker checks for **Memory comparison of padding data**.

Examples

Memory comparison of padding data

Issue

Memory comparison of padding data occurs when you use the `memcmp` function to compare two structures as a whole. In the process, you compare meaningless data stored in the structure padding.

For instance:

```
struct structType {
    char member1;
    int member2;
    .
    .
};

structType var1;
structType var2;
.
.
if(memcmp(&var1,&var2,sizeof(var1)))
{...}
```

Risk

If members of a structure have different data types, your compiler introduces additional padding for data alignment in memory. For an example of padding, see `Higher Estimate of Local Variable Size`.

The content of these extra padding bytes is meaningless. The C Standard allows the content of these bytes to be indeterminate, giving different compilers latitude to implement their own padding. If you perform a byte-by-byte comparison of structures with `memcmp`, you compare even the meaningless data stored in the padding. You might reach the false conclusion that two data structures are not equal, even if their corresponding members have the same value.

Fix

Instead of comparing two structures in one attempt, compare the structures member by member.

For efficient code, write a function that does the comparison member by member. Use this function for comparing two structures.

You can use `memcmp` for byte-by-byte comparison of structures only if you know that the structures do not contain padding. Typically, to prevent padding, you use specific attributes or pragmas such as `#pragma pack`. However, these attributes or pragmas are not supported by all compilers and make your code implementation-dependent. If your structures contain bit-fields, using these attributes or pragmas cannot prevent padding.

Example - Structures Compared with `memcmp`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{

    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    if (0 == memcmp(left, right, sizeof(S_Padding)))
    {
        return 1;
    }
    else
        return 0;
}
```

In this example, `memcmp` compares byte-by-byte the two structures that `left` and `right` point to. Even if the values stored in the structure members are the same, the comparison can show an inequality if the meaningless values in the padding bytes are not the same.

Correction — Compare Structures Member by Member

One possible correction is to compare individual structure members.

Note You can compare entire arrays by using `memcmp`. All members of an array have the same data type. Padding bytes are not required to store arrays.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{
    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    return ((left->c == right->c) &&
            (left->i == right->i) &&
            (left->bf1 == right->bf1) &&
            (left->bf2 == right->bf2) &&
            (memcmp(left->buffer, right->buffer, 20) == 0));
}
```

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

EXP42-C

Introduced in R2019a

CERT C++: EXP45-C

Do not perform assignments in selection statements

Description

Rule Definition

Do not perform assignments in selection statements.

Polyspace Implementation

This checker checks for **Invalid use of = (assignment) operator**.

Examples

Invalid use of = (assignment) operator

Issue

Invalid use of = operator occurs when an assignment is made inside the predicate of a conditional, such as `if` or `while`.

In C and C++, a single equal sign is an assignment not a comparison. Using a single equal sign in a conditional statement can indicate a typo or a mistake.

Risk

- Conditional statement tests the wrong values— The single equal sign operation assigns the value of the right operand to the left operand. Then, because this assignment is inside the predicate of a conditional, the program checks whether the new value of the left operand is nonzero or not NULL.
- Maintenance and readability issues — Even if the assignment is intended, someone reading or updating the code can misinterpret the assignment as an equality comparison instead of an assignment.

Fix

- If the assignment is a bug, to check for equality, add a second equal sign (==).
- If the assignment inside the conditional statement was intentional, to improve readability, separate the assignment and the test. Move the assignment outside the control statement. In the control statement, simply test the result of the assignment.

  If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Single Equal Sign Inside an `if` Condition

```
#include <stdio.h>

void bad_equals_ex(int alpha, int beta)
{
    if(alpha = beta)
```

```
    {
        printf("Equal\n");
    }
}
```

The equal sign is flagged as a defect because the assignment operator is used within the predicate of the if-statement. The predicate assigns the value `beta` to `alpha`, then implicitly tests whether `alpha` is true or false.

Correction — Change Expression to Comparison

One possible correction is adding an additional equal sign. This correction changes the assignment to a comparison. The if condition compares whether `alpha` and `beta` are equal.

```
#include <stdio.h>

void equality_test(int alpha, int beta)
{
    if(alpha == beta)
    {
        printf("Equal\n");
    }
}
```

Correction — Assignment and Comparison Inside the `if` Condition

If an assignment must be made inside the predicate, a possible correction is adding an explicit comparison. This correction assigns the value of `beta` to `alpha`, then explicitly checks whether `alpha` is nonzero. The code is clearer.

```
#include <stdio.h>

int assignment_not_zero(int alpha, int beta)
{
    if((alpha = beta) != 0)
    {
        return alpha;
    }
    else
    {
        return 0;
    }
}
```

Correction — Move Assignment Outside the `if` Statement

If the assignment can be made outside the control statement, one possible correction is to separate the assignment and comparison. This correction assigns the value of `beta` to `alpha` before the if. Inside the if-condition, only `alpha` is given to test if `alpha` is nonzero or not NULL.

```
#include <stdio.h>

void assign_and_print(int alpha, int beta)
{
    alpha = beta;
    if(alpha)
    {
        printf("%d", alpha);
```

```
    }
}
```

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

EXP45-C

Introduced in R2019a

CERT C++: EXP46-C

Do not use a bitwise operator with a Boolean-like operand

Description

Rule Definition

Do not use a bitwise operator with a Boolean-like operand.

Polyspace Implementation

This checker checks for **Use of bool operand with bitwise operator**.

Examples

Use of `bool` operand with bitwise operator

Issue

The issue occurs when you use expressions with type `bool` as operands to built-in operators except for:

- The assignment operator =.
- The logical operators &&, ||, and !.
- The equality operators == and !=.
- The unary operator &.
- The conditional operator.

Risk

Operators other than the ones mentioned in the rule do not produce meaningful results with `bool` operands. Use of `bool` operands with these operators can indicate programming errors. For instance, you intended to use the logical operator `||` but used the bitwise operator `|` instead.

Example - Compliant and Noncompliant Uses of `bool` Operands

```
void boolOperations() {
    bool lhs = true;
    bool rhs = false;

    int res;

    if(lhs & rhs) {} //Noncompliant
    if(lhs < rhs) {} //Noncompliant
    if(~rhs) {}      //Noncompliant
    if(lhs ^ rhs) {} //Noncompliant
    if(lhs == rhs) {} //Compliant
    if(!rhs) {}       //Compliant
    res = lhs? -1:1;  //Compliant
}
```

In this example, `bool` operands do not violate the rule when used with the `==`, `!` and the `?` operators.

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

EXP46-C

Introduced in R2019a

CERT C++: EXP47-C

Do not call va_arg with an argument of the incorrect type

Description

Rule Definition

Do not call va_arg with an argument of the incorrect type.

Polyspace Implementation

This checker checks for these issues:

- **Incorrect data type passed to va_arg**.
- **Too many va_arg calls for current argument list**.

Examples

Incorrect data type passed to va_arg

Issue

Incorrect data type passed to va_arg when the data type in a `va_arg` call does not match the data type of the variadic function argument that `va_arg` reads.

For instance, you pass an `unsigned char` argument to a variadic function `func`. Because of default argument promotion, the argument is promoted to `int`. When you use a `va_arg` call that reads an `unsigned char` argument, a type mismatch occurs.

```
void func (int n, ...) {
   ...
   va_list args;
   va_arg(args, unsigned char);
   ...
}

void main(void) {
   unsigned char c;
   func(1,c);
}
```

Risk

In a variadic function (function with variable number of arguments), you use `va_arg` to read each argument from the variable argument list (`va_list`). The `va_arg` use does not guarantee that there actually exists an argument to read or that the argument data type matches the data type in the `va_arg` call. You have to make sure that both conditions are true.

Reading an incorrect type with a `va_arg` call can result in undefined behavior. Because function arguments reside on the stack, you might access an unwanted area of the stack.

Fix

Make sure that the data type of the argument passed to the variadic function matches the data type in the `va_arg` call.

Arguments of a variadic function undergo default argument promotions. The argument data types of a variadic function cannot be determined from a prototype. The arguments of such functions undergo default argument promotions (see Sec. 6.5.2.2 and 7.15.1.1 in the C99 Standard). Integer arguments undergo integer promotion and arguments of type `float` are promoted to `double`. For integer arguments, if a data type can be represented by an `int`, for instance, `char` or `short`, it is promoted to an `int`. Otherwise, it is promoted to an `unsigned int`. All other arguments do not undergo promotion.

To avoid undefined and implementation-defined behavior, minimize the use of variadic functions. Use the checkers for `MISRA C:2012 Rule 17.1` or `MISRA C++:2008 Rule 8-4-1` to detect use of variadic functions.

Example - `char` Used as Function Argument Type and `va_arg` argument

```
#include <stdarg.h>
#include <stdio.h>

unsigned char func(size_t count, ...) {
    va_list ap;
    unsigned char result = 0;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, unsigned char);
    }
    va_end(ap);
    return result;
}

void func_caller(void) {
    unsigned char c = 0x12;
    (void)func(1, c);
}
```

In this example, `func` takes an `unsigned char` argument, which undergoes default argument promotion to `int`. The data type in the `va_arg` call is still `unsigned char`, which does not match the `int` argument type.

Correction — Use `int` as `va_arg` Argument

One possible correction is to read an `int` argument with `va_arg`.

```
#include <stdarg.h>
#include <stdio.h>

unsigned char func(size_t count, ...) {
    va_list ap;
    unsigned char result = 0;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
    }
    va_end(ap);
    return result;
```

```
}

void func_caller(void) {
    unsigned char c = 0x12;
    (void)func(1, c);
}
```

Too many va_arg calls for current argument list

Issue

Too many va_arg calls for current argument list occurs when the number of calls to `va_arg` exceeds the number of arguments passed to the corresponding variadic function. The analysis raises a defect only when the variadic function is called.

Too many va_arg calls for current argument list does not raise a defect when:

- The number of calls to `va_arg` inside the variadic function is indeterminate. For example, if the calls are from an external source.
- The `va_list` used in `va_arg` is invalid.

Risk

When you call `va_arg` and there is no next argument available in `va_list`, the behavior is undefined. The call to `va_arg` might corrupt data or return an unexpected result.

Fix

Ensure that you pass the correct number of arguments to the variadic function.

Example - No Argument Available When Calling va_arg

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/
int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
        if (count > 0) {
/* No further argument available
* in va_list when calling va_arg
*/

            result += va_arg(ap, int);
        }
    }
    va_end(ap);
    return result;
}
```

```
void func(void) {

    (void)variadic_func(2, 100);

}
```

In this example, the named argument and only one variadic argument are passed to `variadic_func()` when it is called inside `func()`. On the second call to `va_arg`, no further variadic argument is available in `ap` and the behavior is undefined.

Correction — Pass Correct Number of Arguments to Variadic Function

One possible correction is to ensure that you pass the correct number of arguments to the variadic function.

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/

int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
        if (count > 0) {

/* The correct number of arguments is
* passed to va_list when variadic_func()
* is called inside func()
*/
            result += va_arg(ap, int);
        }
    }
    va_end(ap);
    return result;
}

void func(void) {

    (void)variadic_func(2, 100, 200);

}
```

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites
EXP47-C

Introduced in R2019a

CERT C++: EXP50-CPP

Do not depend on the order of evaluation for side effects

Description

Rule Definition

Do not depend on the order of evaluation for side effects.

Polyspace Implementation

This checker checks for **Expression value depends on order of evaluation**.

Examples

Expression value depends on order of evaluation

Issue

The issue occurs when the value of an expression is not the same depending on the order of evaluation of the expression.

An expression can have different values under the following conditions:

- The same variable is modified more than once in the expression, or is both read and written.
- The expression allows more than one order of evaluation.

Therefore, the rule checker forbids expressions where a variable is modified more than once and can cause different results under different orders of evaluation. The rule checker also detects cases where a volatile variable is read more than once in an expression.

Risk

If an expression results in different values depending on the order of evaluation, its value becomes implementation-defined.

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

EXP50-CPP

Introduced in R2019a

CERT C++: EXP52-CPP

Do not rely on side effects in unevaluated operands

Description

Rule Definition

Do not rely on side effects in unevaluated operands.

Polyspace Implementation

This checker checks for **Logical operator operand with side effects**.

Examples

Logical operator operand with side effects

Issue

The issue occurs when the right hand operand of a logical && or || operator contains side effects.

The checker does not show a warning on volatile accesses and function calls.

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

EXP52-CPP

Introduced in R2019a

CERT C++: EXP53-CPP

Do not read uninitialized memory

Description

Rule Definition

Do not read uninitialized memory.

Polyspace Implementation

This checker checks for these issues:

- **Non-initialized pointer**.
- **Non-initialized variable**.

Examples

Non-initialized pointer

Issue

Non-initialized pointer occurs when a pointer is not assigned an address before dereference.

Risk

Unless a pointer is explicitly assigned an address, it points to an unpredictable location.

Fix

The fix depends on the root cause of the defect. For instance, you assigned an address to the pointer but the assignment is unreachable.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below. It is a good practice to initialize a pointer to NULL when declaring the pointer.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Non-initialized pointer error

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;
```

```
    if (prev == NULL)
      {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
      }

    *pi = j;
    /* Defect: Writing to uninitialized pointer */

    return pi;
}
```

If `prev` is not `NULL`, the pointer `pi` is not assigned an address. However, `pi` is dereferenced on every execution paths, irrespective of whether `prev` is `NULL` or not.

Correction — Initialize Pointer on Every Execution Path

One possible correction is to assign an address to `pi` when `prev` is not `NULL`.

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
       {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
       }
    /* Fix: Initialize pi in branches of if statement  */
    else
        pi = prev;


    *pi = j;

    return pi;
}
```

Non-initialized variable

Issue

Non-initialized variable occurs when a variable is not initialized before its value is read.

Risk

Unless a variable is explicitly initialized, the variable value is unpredictable. You cannot rely on the variable having a specific value.

Fix

The fix depends on the root cause of the defect. For instance, you assigned a value to the variable but the assignment is unreachable or you assigned a value to the variable in one of two branches of a conditional statement. Fix the unreachable code or missing assignment.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back

using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below. It is a good practice to initialize a variable at declaration.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Non-initialized variable error

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    int val;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
    /* Defect: val does not have a value if command is not 2 */
}
```

If `command` is not 2, the variable `val` is unassigned. In this case, the return value of function `get_sensor_value` is undetermined.

Correction — Initialize During Declaration

One possible correction is to initialize `val` during declaration so that the initialization is not bypassed on some execution paths.

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    /* Fix: Initialize val */
    int val=0;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
 }
```

`val` is assigned an initial value of 0. When `command` is not equal to 2, the function `get_sensor_value` returns this value.

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

EXP53-CPP

Introduced in R2019a

CERT C++: EXP54-CPP

Do not access an object outside of its lifetime

Description

Rule Definition

Do not access an object outside of its lifetime.

Polyspace Implementation

This checker checks for these issues:

- **Non-initialized pointer**.
- **Non-initialized variable**.
- **Use of previously freed pointer**.
- **Pointer or reference to stack variable leaving scope**.
- **Accessing object with temporary lifetime**.

Examples

Non-initialized pointer

Issue

Non-initialized pointer occurs when a pointer is not assigned an address before dereference.

Risk

Unless a pointer is explicitly assigned an address, it points to an unpredictable location.

Fix

The fix depends on the root cause of the defect. For instance, you assigned an address to the pointer but the assignment is unreachable.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below. It is a good practice to initialize a pointer to NULL when declaring the pointer.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Non-initialized pointer error

```
#include <stdlib.h>

int* assign_pointer(int* prev)
```

```
{
    int j = 42;
    int* pi;

    if (prev == NULL)
      {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
      }

    *pi = j;
    /* Defect: Writing to uninitialized pointer */

    return pi;
}
```

If `prev` is not `NULL`, the pointer `pi` is not assigned an address. However, `pi` is dereferenced on every execution paths, irrespective of whether `prev` is `NULL` or not.

Correction — Initialize Pointer on Every Execution Path

One possible correction is to assign an address to `pi` when `prev` is not `NULL`.

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
       {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
       }
    /* Fix: Initialize pi in branches of if statement  */
    else
        pi = prev;


    *pi = j;

    return pi;
}
```

Non-initialized variable

Issue

Non-initialized variable occurs when a variable is not initialized before its value is read.

Risk

Unless a variable is explicitly initialized, the variable value is unpredictable. You cannot rely on the variable having a specific value.

Fix

The fix depends on the root cause of the defect. For instance, you assigned a value to the variable but the assignment is unreachable or you assigned a value to the variable in one of two branches of a conditional statement. Fix the unreachable code or missing assignment.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below. It is a good practice to initialize a variable at declaration.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Non-initialized variable error

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    int val;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
    /* Defect: val does not have a value if command is not 2 */
}
```

If `command` is not 2, the variable `val` is unassigned. In this case, the return value of function `get_sensor_value` is undetermined.

Correction — Initialize During Declaration

One possible correction is to initialize `val` during declaration so that the initialization is not bypassed on some execution paths.

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    /* Fix: Initialize val */
    int val=0;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
 }
```

`val` is assigned an initial value of 0. When `command` is not equal to 2, the function `get_sensor_value` returns this value.

Use of previously freed pointer

Issue

Use of previously freed pointer occurs when you access a block of memory after freeing the block using the `free` function.

Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to access this block of memory can result in unpredictable behavior or even a segmentation fault.

Fix

The fix depends on the root cause of the defect. See if you intended to free the memory later or allocate another memory block to the pointer before access.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before dereferencing pointers, check them for NULL values and handle the error. In this way, you are protected against accessing a freed block.

Example - Use of Previously Freed Pointer Error

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   {
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);

    j = *pi + shift;
    /* Defect: Reading a freed pointer */

    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, dereferencing`pi` after the `free` statement is not valid.

Correction — Free Pointer After Use

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
#include <stdlib.h>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;
```

```
    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

Pointer or reference to stack variable leaving scope

Issue

Pointer or reference to stack variable leaving scope occurs when a pointer or reference to a local variable leaves the scope of the variable. For instance:

- A function returns a pointer to a local variable.
- A function performs the assignment `globPtr = &locVar`. `globPtr` is a global pointer variable and `locVar` is a local variable.
- A function performs the assignment `*paramPtr = &locVar`. `paramPtr` is a function parameter that is, for instance, an `int**` pointer and `locVar` is a local `int` variable.
- A C++ method performs the assignment `memPtr = &locVar`. `memPtr` is a pointer data member of the class the method belongs to. `locVar` is a variable local to the method.

The defect also applies to memory allocated using the `alloca` function. The defect does not apply to static, local variables.

Risk

Local variables are allocated an address on the stack. Once the scope of a local variable ends, this address is available for reuse. Using this address to access the local variable value outside the variable scope can cause unexpected behavior.

If a pointer to a local variable leaves the scope of the variable, Polyspace Bug Finder highlights the defect. The defect appears even if you do not use the address stored in the pointer. For maintainable code, it is a good practice to not allow the pointer to leave the variable scope. Even if you do not use the address in the pointer now, someone else using your function can use the address, causing undefined behavior.

Fix

Do not allow a pointer or reference to a local variable to leave the variable scope.

Example - Pointer to Local Variable Returned from Function

```
void func2(int *ptr) {
    *ptr = 0;
}

int* func1(void) {
    int ret = 0;
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
```

```
    func2(ptr) ;
}
```

In this example, `func1` returns a pointer to local variable `ret`.

In `main`, `ptr` points to the address of the local variable. When `ptr` is accessed in `func2`, the access is illegal because the scope of `ret` is limited to `func1`,

Example - Pointer to Local Variable Escapes Through Lambda Expression

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [&] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

In this example, the `createAdder` function defines a lambda expression `adder` that captures the local variable `addThis` by reference. The scope of `addThis` is limited to the `createAdder` function. When the object returned by `createAdder` is called, a reference to the variable `addThis` is accessed outside its scope. When accessed in this way, the value of `addThis` is undefined.

Correction – Capture Local Variables by Copy in Lambda Expression Instead of Reference

If a function returns a lambda expression object, avoid capturing local variables by reference in the lambda object. Capture the variables by copy instead.

Variables captured by copy have the same lifetime as the lambda object, but variables captured by reference often have a smaller lifetime than the lambda object itself. When the lambda object is used, these variables accessed outside scope have undefined values.

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [=] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

Accessing object with temporary lifetime

Issue

Accessing object with temporary lifetime occurs when you attempt to read from or write to an object with temporary lifetime that is returned by a function call. In a structure or union returned by a function, and containing an array, the array members are temporary objects. The lifetime of temporary objects ends:

- When the full expression or full declarator containing the call ends, as defined in the C11 Standard.
- After the next sequence point, as defined in the C90 and C99 Standards. A sequence point is a point in the execution of a program where all previous evaluations are complete and no subsequent evaluation has started yet.

For C++ code, **Accessing object with temporary lifetime** raises a defect only when you write to an object with a temporary lifetime.

If the temporary lifetime object is returned by address, no defect is raised.

Risk

Modifying objects with temporary lifetime is undefined behavior and can cause abnormal program termination and portability issues.

Fix

Assign the object returned from the function call to a local variable. The content of the temporary lifetime object is copied to the variable. You can now modify it safely.

Example - Modifying Temporary Lifetime Object Returned by Function Call

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6

struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

/* func_temp() returns a struct value containing
* an array with a temporary lifetime.
*/
int func(void) {

/*Writing to temporary lifetime object is
 undefined behavior
 */
    return ++(func_temp().a[0]);
}

void main(void) {
    (void)func();
}
```

In this example, `func_temp()` returns by value a structure with an array member `a`. This member has temporary lifetime. Incrementing it is undefined behavior.

Correction — Assign Returned Value to Local Variable Before Writing

One possible correction is to assign the return of the call to `func_temp()` to a local variable. The content of the temporary object `a` is copied to the variable, which you can safely increment.

```
 #include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6

struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

int func(void) {

/* Assign object returned by function call to
 *local variable
 */
    struct S_Array s = func_temp();

/* Local variable can safely be
 *incremented
 */
    ++(s.a[0]);
    return s.a[0];
}

void main(void) {
    (void)func();
}
```

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

EXP54-CPP

Introduced in R2019a

CERT C++: EXP55-CPP

Do not access a cv-qualified object through a cv-unqualified type

Description

Rule Definition

Do not access a cv-qualified object through a cv-unqualified type.

Polyspace Implementation

This checker checks for **Cast removes cv-qualification of pointer**.

Examples

Cast removes cv-qualification of pointer

Issue

The issue occurs when a cast removes a `const` or `volatile` qualification from the type of a pointer or reference.

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics
“Check for Coding Standard Violations”

External Websites
EXP55-CPP

Introduced in R2019a

CERT C++: EXP57-CPP

Do not cast or delete pointers to incomplete classes

Description

Rule Definition

Do not cast or delete pointers to incomplete classes.

Polyspace Implementation

This checker checks for **Conversion or deletion of incomplete class pointer**.

Examples

Conversion or deletion of incomplete class pointer

Issue

Conversion or deletion of incomplete class pointer occurs when you delete or cast to a pointer to an incomplete class. An incomplete class is one whose definition is not visible at the point where the class is used.

For instance, the definition of class `Body` is not visible when the `delete` operator is called on a pointer to `Body`:

```
class Handle {
  class Body *impl;
public:
  ~Handle() { delete impl; }
  // ...
};
```

Risk

When you delete a pointer to an incomplete class, it is not possible to call any nontrivial destructor that the class might have. If the destructor performs cleanup activities such as memory deallocation, these activities do not happen.

A similar problem happens, for instance, when you downcast to a pointer to an incomplete class (downcasting is casting from a pointer to a base class to a pointer to a derived class). At the point of downcasting, the relationship between the base and derived class is not known. In particular, if the derived class inherits from multiple classes, at the point of downcasting, this information is not available. The downcasting cannot make the necessary adjustments for multiple inheritance and the resulting pointer cannot be dereferenced.

A similar statement can be made for upcasting (casting from a pointer to derived class to a pointer to a base class).

Fix

When you delete or downcast to a pointer to a class, make sure that the class definition is visible.

Alternatively, you can perform one of these actions:

- Instead of a regular pointer, use the `std::shared_ptr` type to point to the incomplete class.
- When downcasting, make sure that the result is valid. Write error-handling code for invalid results.

Example - Deletion of Pointer to Incomplete Class

```
class Handle {
  class Body *impl;
public:
  ~Handle() { delete impl; }
  // ...
};
```

In this example, the definition of class `Body` is not visible when the pointer to `Body` is deleted.

Correction — Define Class Before Deletion

One possible correction is to make sure that the class definition is visible when a pointer to the class is deleted.

```
class Handle {
  class Body *impl;
public:
  ~Handle();
  // ...
};

// Elsewhere
class Body { /* ... */ };

Handle::~Handle() {
  delete impl;
}
```

Correction — Use `std::shared_ptr`

Another possible correction is to use the `std::shared_ptr` type instead of a regular pointer.

```
#include <memory>

class Handle {
  std::shared_ptr<class Body> impl;
  public:
    Handle();
    ~Handle() {}
    // ...
};
```

Example - Downcasting to Pointer to Incomplete Class

`File1.h`:

```
class Base {
protected:
  double var;
public:
  Base() : var(1.0) {}
```

```
  virtual void do_something();
  virtual ~Base();
};
```

`File2.h`:

```
void funcprint(class Derived *);
class Base *get_derived();
```

`File1.cpp`:

```
#include "File1.h"
#include "File2.h"

void getandprint() {
  Base *v = get_derived();
  funcprint(reinterpret_cast<class Derived *>(v));
}
```

`File2.cpp`:

```
#include "File2.h"
#include "File1.h"
#include <iostream>

class Base2 {
protected:
  short var2;
public:
  Base2() : var2(12) {}
};

class Derived : public Base2, public Base {
  float var_derived;
public:
    Derived() : Base2(), Base(), var_derived(1.2f) {}
    void do_something()
    {
        std::cout << "var_derived: "
                  << var_derived << ", var : " << var
                  << ", var2: " << var2 << std::endl;
    }
 };

void funcprint(Derived *d) {
  d->do_something();
}

Base *get_derived() {
  return new Derived;
}
```

In this example, the definition of class `Derived` is not visible in `File1.cpp` when a `Base*` pointer to downcast to a `Derived*` pointer.

In `File2.cpp`, class `Derived` derives from two classes, `Base` and `Base2`. This information about multiple inheritance is not available at the point of downcasting in `File1.cpp`. The result of

downcasting is passed to the function `funcprint` and dereferenced in the body of `funcprint`. Because the downcasting was done with incomplete information, the dereference can be invalid.

Correction — Define Class Before Downcasting

One possible correction is to define the class `Derived` before downcasting a `Base*` pointer to a `Derived*` pointer.

In this corrected example, the downcasting is done in `File2.cpp` in the body of `funcprint` at a point where the definition of class `Derived` is visible. The downcasting is not done in `File1.cpp` where the definition of `Derived` is not visible. The changes from the previous incorrect example are highlighted.

`File1.h`:

```
class Base {
protected:
  double var;
public:
  Base() : var(1.0) {}
  virtual void do_something();
  virtual ~Base();
};
```

`File2.h`:

```
void funcprint(class Base *);
class Base *get_derived();
```

`File1.cpp`:

```
#include "File1.h"
#include "File2.h"

void getandprint() {
  Base *v = get_derived();
  funcprint(v);
}
```

`File2.cpp`:

```
#include "File2_corr.h"
#include "File1_corr.h"
#include <iostream>

class Base2 {
protected:
  short var2;
public:
  Base2() : var2(12) {}
};

class Derived : public Base2, public Base {
  float var_derived;

public:
    Derived() : Base2(), Base(), var_derived(1.2f) {}
    void do_something()
```

```
    {
        std::cout << "var_derived: "
                  << var_derived << ", var : " << var
                  << ", var2: " << var2 << std::endl;
    }
};

void funcprint(Base *d) {
  Derived *temp = dynamic_cast<Derived*>(d);
  if(temp)  {
     d->do_something();
  }
  else {
      //Handle error
  }
}

Base *get_derived() {
  return new Derived;
}
```

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

EXP57-CPP

Introduced in R2019a

CERT C++: EXP58-CPP

Pass an object of the correct type to va_start

Description

Rule Definition

Pass an object of the correct type to va_start.

Polyspace Implementation

This checker checks for **Incorrect Data Types for Second Argument of va_start**.

Examples

Incorrect Data Types for Second Argument of `va_start`

```
#include <string>
#include <cstdarg>

double addVariableNumberOfDoubles(double* weight, short num, ...) {
    double sum=0.0;
    va_list list;
    va_start(list, num);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, double);
    }
    va_end(list);
    return sum;
}

double addVariableNumberOfFloats(float* weight, int num, std::string s, ...) {
    float sum=0.0;
    va_list list;
    va_start(list, s);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, float);
    }
    va_end(list);
    return sum;
}
```

In this example, the checker flags the call to `va_start` in:

- `addVariableNumberOfDoubles` because the argument has type `short`, which undergoes default argument promotion to `int`.
- `addVariableNumberOfFloats` because the argument has type `std::string`, which has a nontrivial copy constructor.

Issue

Make sure that the second argument of the `va_start` macro has a supported data type. In the following corrected example:

- In `addVariableNumberOfDoubles`, the data type of the last named parameter of the variadic function is changed to `int`.
- In `addVariableNumberOfFloats`, the second and third parameters of the variadic function are switched so that data type of the last named parameter is `int`.

```
#include <string>
#include <cstdarg>

double addVariableNumberOfDoubles(double* weight, int num, ...) {
    double sum=0.0;
    va_list list;
    va_start(list, num);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, double);
    }
    va_end(list);
    return sum;
}

double addVariableNumberOfFloats(double* weight, std::string s, int num, ...) {
    double sum=0.0;
    va_list list;
    va_start(list, num);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, double);
    }
    va_end(list);
    return sum;
}
```

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

EXP58-CPP

Introduced in R2019a

CERT C++: EXP59-CPP

Use offsetof() on valid types and members

Description

Rule Definition

Use offsetof() on valid types and members.

Polyspace Implementation

This checker checks for **Use of offsetof Macro with Nonstandard Layout Class**.

Examples

Use of `offsetof` Macro with Nonstandard Layout Class

```
#include <cstddef>

class myClass {
     int privateData;
  public:
     int publicData;
};

void func() {
  size_t off = offsetof(myClass, publicData);
  // ...
}
```

In this example, the class `myClass` has two data members with different access control, one private and the other public. Therefore, the class does not satisfy the requirements of a standard layout class and cannot be used with the `offsetof` macro.

Issue

If the use of `offsetof` is important for the application, make sure that the first argument is a class with a standard layout. For instance, see if you can work around the need for a public data member.

```
#include <cstddef>

class myClass {
     int privateData;
     int publicData;
  public:
     int getpublicData(void) { return publicData;}
};

void func() {
  size_t off = offsetof(myClass, publicData);
  // ...
}
```

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

EXP59-CPP

Introduced in R2019a

CERT C++: EXP61-CPP

A lambda object must not outlive any of its reference captured objects

Description

Rule Definition

A lambda object must not outlive any of its reference captured objects.

Polyspace Implementation

This checker checks for **Object Escapes Scope Through Lambda Expression**.

Examples

Object Escapes Scope Through Lambda Expression

Issue

The issue occurs when a lambda expression captures an object *by reference* and the lambda expression object outlives the captured object. For instance, the captured object is a local variable but the lambda expression object has a much larger scope.

Risk

If a lambda expression object outlives one of its reference captured objects, the captured object can be accessed outside its scope.

For instance, consider this function `createFunction`:

```
std::function<std::int32_t()> createFunction() {
   std::int32_t localVar = 0;
   return ([&localVar]() -> std::int32_t {
       localVar = 1;
       return localVar;
   });
}
```

`createFunction` returns a lambda expression object that captures the local variable `localVar` *by reference*. The scope of `localVar` is limited to `createFunction` but the lambda expression object returned has a much larger scope.

This situation can result in an attempt to access the local object `localVar` outside its scope. For instance, when you call `createFunction` and assign the returned lambda expression object to another object `aFunction`:

```
auto aFunction = createFunction();
```

and then invoke the new object `aFunction`:

```
std::int32_t someValue = aFunction();
```

the captured variable `localVar` is no longer in scope. Therefore, the value returned from `aFunction` is undefined.

Fix

If a function returns a lambda expression, to avoid accessing a captured object outside its scope, make sure that the lambda expression captures all objects by copy. For instance, you can rewrite `createFunction` as:

```
std::function<std::int32_t()> createFunction() {
   std::int32_t localVar = 0;
   return ([localVar]() mutable -> std::int32_t {
       localVar = 1;
       return localVar;
   });
}
```

Example – Pointer to Local Variable Escapes Through Lambda Expression

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [&] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

In this example, the `createAdder` function defines a lambda expression `adder` that captures the local variable `addThis` by reference. The scope of `addThis` is limited to the `createAdder` function. When the object returned by `createAdder` is called, a reference to the variable `addThis` is accessed outside its scope. When accessed in this way, the value of `addThis` is undefined.

Correction – Capture Local Variables by Copy in Lambda Expression Instead of Reference

If a function returns a lambda expression object, avoid capturing local variables by reference in the lambda object. Capture the variables by copy instead.

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [=] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

Check Information

Group: 02. Expressions (EXP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics
“Check for Coding Standard Violations”

External Websites
EXP61-CPP

Introduced in R2019b

CERT C++: INT30-C

Ensure that unsigned integer operations do not wrap

Description

Rule Definition

Ensure that unsigned integer operations do not wrap.

Polyspace Implementation

This checker checks for these issues:

- **Unsigned integer overflow**.
- **Unsigned integer constant overflow**.

Examples

Unsigned integer overflow

Issue

Unsigned integer overflow occurs when an operation on unsigned integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

The C11 standard states that unsigned integer overflows result in wrap-around behavior. However, a wrap around behavior might not always be desirable. For instance, if the result of a computation is used as an array size and the computation overflows, the array size is much smaller than expected.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling. In the error handling code, you can override the default wrap-around behavior for overflows and implement saturation behavior, for instance.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Add One to Maximum Unsigned Integer

```
#include <limits.h>

unsigned int plusplus(void) {

    unsigned uvar = UINT_MAX;
    uvar++;
    return uvar;
}
```

In the third statement of this function, the variable `uvar` is increased by 1. However, the value of `uvar` is the maximum unsigned integer value, so 1 plus the maximum integer value cannot be represented by an `unsigned int`. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `uvar` is reduced by modulo `UINT_MAX`. The result is `uvar = 1`.

Correction — Different Storage Type

One possible correction is to store the operation result in a larger data type. In this example, by returning an `unsigned long long` instead of an `unsigned int`, the overflow error is fixed.

```
#include <limits.h>

unsigned long long plusplus(void) {

    unsigned long long ullvar = UINT_MAX;
    ullvar++;
    return ullvar;
}
```

Unsigned integer constant overflow

Issue

Unsigned integer constant overflow occurs when you assign a compile-time constant to a unsigned integer variable whose data type cannot accommodate the value. An `n`-bit unsigned integer holds values in the range `[0, 2`^{n}`-1]`.

For instance, `c` is an 8-bit unsigned `char` variable that cannot hold the value 256.

```
unsigned char c = 256;
```

To determine the sizes of fundamental types, Bug Finder uses your specification for `Target processor type (-target)`.

Risk

The C standard states that overflowing unsigned integers must be wrapped around (see, for instance, the C11 standard, section 6.2.5). However, the wrap-around behavior can be unintended and cause unexpected results.

Fix

Check if the constant value is what you intended. If the value is correct, use a wider data type for the variable.

Example - Overflowing Constant from Macro Expansion

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_UNSIGNED_SHORT 65535

void main() {
    unsigned char c1 = MAX_UNSIGNED_CHAR + 1;
    unsigned short c2 = MAX_UNSIGNED_SHORT + 1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow.

Correction — Use Wider Data Type

One possible correction is to use a wider data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_UNSIGNED_SHORT 65535

void main() {
    unsigned short c1 = MAX_UNSIGNED_CHAR + 1;
    unsigned int c2 = MAX_UNSIGNED_SHORT + 1;
}
```

Check Information

Group: 03. Integers (INT)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

INT30-C

Introduced in R2019a

CERT C++: INT31-C

Ensure that integer conversions do not result in lost or misinterpreted data

Description

Rule Definition

Ensure that integer conversions do not result in lost or misinterpreted data.

Polyspace Implementation

This checker checks for these issues:

- **Integer conversion overflow**.
- **Call to memset with unintended value**.
- **Sign change integer conversion overflow**.
- **Tainted sign change conversion**.
- **Unsigned integer conversion overflow**.

Examples

Integer conversion overflow

Issue

Integer conversion overflow occurs when converting an integer to a smaller integer type. If the variable does not have enough bytes to represent the original value, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Integer conversion overflows result in undefined behavior.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Converting from `int` to `char`

```
char convert(void) {

    int num = 1000000;

    return (char)num;
}
```

In the return statement, the integer variable `num` is converted to a `char`. However, an 8-bit or 16-bit character cannot represent 1000000 because it requires at least 20 bits. So the conversion operation overflows.

Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number.

```
long convert(void) {

    int num = 1000000;

    return (long)num;
}
```

Call to memset with unintended value

Issue

Call to memset with unintended value occurs when Polyspace Bug Finder detects a use of the `memset` or `wmemset` function with possibly incorrect arguments.

`void *memset (void *ptr, int value, size_t num)` fills the first `num` bytes of the memory block that `ptr` points to with the specified `value`. If the argument `value` is incorrect, the memory block is initialized with an unintended value.

The unintended initialization can occur in the following cases.

Issue	Risk	Possible Fix
The second argument is `'0'` instead of `0` or `'\0'`.	The ASCII value of character `'0'` is 48 (decimal), `0x30` (hexadecimal), `069` (octal) but not `0` (or `'\0'`) .	If you want to initialize with `'0'`, use one of the ASCII values. Otherwise, use `0` or `'\0'`.
The second and third arguments are probably reversed. For instance, the third argument is a literal and the second argument is not a literal.	If the order is reversed, a memory block of unintended size is initialized with incorrect arguments.	Reverse the order of the arguments.

Issue	Risk	Possible Fix
The second argument cannot be represented in a byte.	If the second argument cannot be represented in a byte, and you expect each byte of a memory block to be filled with that argument, the initialization does not occur as intended.	Apply a bit mask to the argument to produce a wrapped or truncated result that can be represented in a byte. When you apply a bit mask, make sure that it produces an expected result. For instance, replace `memset(a, -13, sizeof(a))` with `memset(a, (-13) & 0xFF, sizeof(a))`.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Value Cannot Be Represented in a Byte

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE];
    int c = -2;
    memset(buf, (char)c, sizeof(buf));
}
```

In this example, `(char)c` cannot be represented in a byte.

Correction — Apply Cast

One possible correction is to apply a cast so that the result can be represented in a byte. However, check that the result of the cast is an acceptable initialization value.

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE   ];
    int c = -2;
    memset(buf, (unsigned char)c, sizeof(buf));
}
```

Sign change integer conversion overflow

Issue

Sign change integer conversion overflow occurs when converting an unsigned integer to a signed integer. If the variable does not have enough bytes to represent both the original constant and the sign bit, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Convert from `unsigned char` to `char`

```
char sign_change(void) {
    unsigned char count = 255;

    return (char)count;
}
```

In the return statement, the unsigned character variable `count` is converted to a signed character. However, `char` has 8 bits, 1 for the sign of the constant and 7 to represent the number. The conversion operation overflows because 255 uses 8 bits.

Correction — Change conversion types

One possible correction is using a larger integer type. By using an `int`, there are enough bits to represent the sign and the number value.

```
int sign_change(void) {
    unsigned char count = 255;

    return (int)count;
}
```

Tainted sign change conversion

Issue

Tainted sign change conversion looks for values from unsecure sources that are converted, implicitly or explicitly, from signed to unsigned values.

For example, functions that use `size_t` as arguments implicitly convert the argument to an unsigned integer. Some functions that implicitly convert `size_t` are:

```
bcmp
memcpy
```

```
memmove
strncmp
strncpy
calloc
malloc
memalign
```

Risk

If you convert a small negative number to unsigned, the result is a large positive number. The large positive number can create security vulnerabilities. For example, if you use the unsigned value in:

- Memory size routines — causes allocating memory issues.
- String manipulation routines — causes buffer overflow.
- Loop boundaries — causes infinite loops.

Fix

To avoid converting unsigned negative values, check that the value being converted is within an acceptable range. For example, if the value represents a size, validate that the value is not negative and less than the maximum value size.

Example - Set Memory Value with Size Argument

```
#include <stdlib.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedsignchange(int size) {
    char str[SIZE128] = "";
    if (size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

In this example, a `char` buffer is created and filled using `memset`. The size argument to `memset` is an input argument to the function.

The call to `memset` implicitly converts `size` to unsigned integer. If `size` is a large negative number, the absolute value could be too large to represent as an integer, causing a buffer overflow.

Correction — Check Value of `size`

One possible correction is to check if `size` is inside the valid range. This correction checks if `size` is greater than zero and less than the buffer size before calling `memset`.

```
#include <stdlib.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
```

```
};

void corrected_taintedsignchange(int size) {
    char str[SIZE128] = "";
    if (size>0 && size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

Unsigned integer conversion overflow

Issue

Unsigned integer conversion overflow occurs when converting an unsigned integer to a smaller unsigned integer type. If the variable does not have enough bytes to represent the original constant, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Integer conversion overflows result in undefined behavior.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Converting from `int` to `char`

```
unsigned char convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned char)unum;
}
```

In the return statement, the unsigned integer variable `unum` is converted to an unsigned character type. However, the conversion overflows because 1000000 requires at least 20 bits. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `unum` is reduced by modulo `2^8` because a character data type can only represent `2^8-1`.

Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number. For example, `long`.

```
unsigned long convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned long)unum;
}
```

Check Information

Group: 03. Integers (INT)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

INT31-C

Introduced in R2019a

CERT C++: INT32-C

Ensure that operations on signed integers do not result in overflow

Description

Rule Definition

Ensure that operations on signed integers do not result in overflow.

Polyspace Implementation

This checker checks for these issues:

- **Integer overflow**.
- **Tainted division operand**.
- **Tainted modulo operand**.

Examples

Integer overflow

Issue

Integer overflow occurs when an operation on integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Integer overflows on signed integers result in undefined behavior.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

To avoid overflows in general, try one of these techniques:

- Keep integer variable values restricted to within half the range of signed integers.

- In operations that might overflow, check for conditions that can lead to the overflow and implement wrap around or saturation behavior depending on how the result of the operation is used. The result then becomes predictable and can be safely used in subsequent computations.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Addition of Maximum Integer

```
#include <limits.h>

int plusplus(void) {

    int var = INT_MAX;
    var++;
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

Correction — Different Storage Type

One possible correction is to change data types. Store the result of the operation in a larger data type (Note that on a 32-bit machine, `int` and `long` has the same size). In this example, on a 32-bit machine, by returning a `long long` instead of an `int`, the overflow error is fixed.

```
#include <limits.h>

long long plusplus(void) {

    long long lvar = INT_MAX;
    lvar++;
    return lvar;
}
```

Tainted division operand

Issue

Tainted division operand detects division operations where one or both of the integer operands is from an unsecure source.

Risk

- If the numerator is the minimum possible value and the denominator is `-1`, your division operation overflows because the result cannot be represented by the current variable size.
- If the denominator is zero, your division operation fails possibly causing your program to crash.

These risks can be used to execute arbitrary code. This code is usually outside the scope of a program's implicit security policy.

Fix

Before performing the division, validate the values of the operands. Check for denominators of `0` or `-1`, and numerators of the minimum integer value.

Example - Division of Function Arguments

```
extern void print_int(int);

int taintedintdivision(int usernum, int userden) {
    int r =  usernum/userden;
    print_int(r);
    return r;
}
```

This example function divides two argument variables, then prints and returns the result. The argument values are unknown and can cause division by zero or integer overflow.

Correction — Check Values

One possible correction is to check the values of the numerator and denominator before performing the division.

```
#include "limits.h"

extern void print_int(int);

int taintedintdivision(int usernum, int userden) {
    int r = 0;
    if (userden!=0 && !(usernum=INT_MIN && userden==-1)) {
        r = usernum/userden;
    }
    print_int(r);
    return r;
}
```

Tainted modulo operand

Issue

Tainted modulo operand checks the operands of remainder `%` operations. Bug Finder flags modulo operations with one or more tainted operands.

Risk

- If the second remainder operand is zero, your remainder operation fails, causing your program to crash.
- If the second remainder operand is `-1`, your remainder operation can overflow if the remainder operation is implemented based on the division operation that can overflow.
- If one of the operands is negative, the operation result is uncertain. For C89, the modulo operation is not standardized, so the result from negative operands is implementation-defined.

These risks can be exploited by attackers to gain access to your program or the target in general.

Fix

Before performing the modulo operation, validate the values of the operands. Check the second operand for values of `0` and `-1`. Check both operands for negative values.

Example - Modulo with Function Arguments

```
extern void print_int(int);
```

```
int taintedintmod(int userden) {
    int rem =  128%userden;
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using an input argument. The argument is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

Correction — Check Operand Values

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
extern void print_int(int);

int taintedintmod(int userden) {
    int rem = 0;
    if (userden > 0) {
        rem = 128 % userden;
    }
    print_int(rem);
    return rem;
}
```

Check Information

Group: 03. Integers (INT)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

INT32-C

Introduced in R2019a

CERT C++: INT33-C

Ensure that division and remainder operations do not result in divide-by-zero errors

Description

Rule Definition

Ensure that division and remainder operations do not result in divide-by-zero errors.

Polyspace Implementation

This checker checks for these issues:

- **Integer division by zero**.
- **Tainted division operand**.
- **Tainted modulo operand**.

Examples

Integer division by zero

Issue

Integer division by zero occurs when the denominator of a division or modulo operation can be a zero-valued integer.

Risk

A division by zero can result in a program crash.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the denominator variable acquires a zero value. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

It is a good practice to check for zero values of a denominator before division and handle the error. Instead of performing the division directly:

```
res = num/den;
```

use a library function that handles zero values of the denominator before performing the division:

```
res = div(num, den);
```

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Dividing an Integer by Zero

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    result = num/denom;

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

Correction — Check Before Division

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    if (denom != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
int fraction(int num)
{
    int denom = 2;
    int result = 0;

    result = num/denom;

    return result;
}
```

Example - Modulo Operation with Zero

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        arr[i] = input % i;
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

In this example, Polyspace flags the modulo operation as a division by zero. Because modulo is inherently a division operation, the divisor (right hand argument) cannot be zero. The modulo

operation uses the `for` loop index as the divisor. However, the `for` loop starts at zero, which cannot be an iterator.

Correction — Check Divisor Before Operation

One possible correction is checking the divisor before the modulo operation. In this example, see if the index `i` is zero before the modulo operation.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        if(i != 0)
        {
             arr[i] = input % i;
        }
        else
        {
             arr[i] = input;
        }
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

Correction — Change Divisor

Another possible correction is changing the divisor to a nonzero integer. In this example, add one to the index before the `%` operation to avoid dividing by zero.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
         arr[i] = input % (i+1);
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

Tainted division operand

Issue

Tainted division operand detects division operations where one or both of the integer operands is from an unsecure source.

Risk

- If the numerator is the minimum possible value and the denominator is `-1`, your division operation overflows because the result cannot be represented by the current variable size.
- If the denominator is zero, your division operation fails possibly causing your program to crash.

These risks can be used to execute arbitrary code. This code is usually outside the scope of a program's implicit security policy.

Fix

Before performing the division, validate the values of the operands. Check for denominators of `0` or `-1`, and numerators of the minimum integer value.

Example - Division of Function Arguments

```
extern void print_int(int);

int taintedintdivision(int usernum, int userden) {
    int r =  usernum/userden;
    print_int(r);
    return r;
}
```

This example function divides two argument variables, then prints and returns the result. The argument values are unknown and can cause division by zero or integer overflow.

Correction — Check Values

One possible correction is to check the values of the numerator and denominator before performing the division.

```
#include "limits.h"

extern void print_int(int);

int taintedintdivision(int usernum, int userden) {
    int r = 0;
    if (userden!=0 && !(usernum=INT_MIN && userden==-1)) {
        r = usernum/userden;
    }
    print_int(r);
    return r;
}
```

Tainted modulo operand

Issue

Tainted modulo operand checks the operands of remainder `%` operations. Bug Finder flags modulo operations with one or more tainted operands.

Risk

- If the second remainder operand is zero, your remainder operation fails, causing your program to crash.
- If the second remainder operand is `-1`, your remainder operation can overflow if the remainder operation is implemented based on the division operation that can overflow.
- If one of the operands is negative, the operation result is uncertain. For C89, the modulo operation is not standardized, so the result from negative operands is implementation-defined.

These risks can be exploited by attackers to gain access to your program or the target in general.

Fix

Before performing the modulo operation, validate the values of the operands. Check the second operand for values of `0` and `-1`. Check both operands for negative values.

Example - Modulo with Function Arguments

```
extern void print_int(int);

int taintedintmod(int userden) {
    int rem =  128%userden;
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using an input argument. The argument is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

Correction — Check Operand Values

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
extern void print_int(int);

int taintedintmod(int userden) {
    int rem = 0;
    if (userden > 0) {
        rem = 128 % userden;
    }
    print_int(rem);
    return rem;
}
```

Check Information

Group: 03. Integers (INT)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

INT33-C

Introduced in R2019a

CERT C++: INT34-C

Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand

Description

Rule Definition

Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand.

Polyspace Implementation

This checker checks for these issues:

- **Shift of a negative value**.
- **Shift operation overflow**.

Examples

Shift of a negative value

Issue

Shift of a negative value occurs when a bit-wise shift is used on a variable that can have negative values.

Risk

Shifts on negative values overwrite the sign bit that identifies a number as negative. The shift operation can result in unexpected values.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variable being shifted acquires negative values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

To fix the defect, check for negative values before the bit-wise shift operation and perform appropriate error handling.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Shifting a negative variable

```
int shifting(int val)
{
    int res = -1;
```

```
    return res << val;
}
```

In the return statement, the variable `res` is shifted a certain number of bits to the left. However, because `res` is negative, the shift might overwrite the sign bit.

Correction — Change the Data Type

One possible correction is to change the data type of the shifted variable to unsigned. This correction eliminates the sign bit, so left shifting does not change the sign of the variable.

```
int shifting(int val)
{
    unsigned int res = -1;
    return res << val;
}
```

Shift operation overflow

Issue

Shift operation overflow occurs when a shift operation can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Shift operation overflows can result in undefined behavior.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the shift operation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the shift operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Left Shift of Integer

```
int left_shift(void) {

    int foo = 33;
    return 1 << foo;
}
```

In the return statement of this function, bit-wise shift operation is performed shifting 1 `foo` bits to the left. However, an `int` has only 32 bits, so the range of the shift must be between 0 and 31. Therefore, this shift operation causes an overflow.

Correction — Different storage type

One possible correction is to store the shift operation result in a larger data type. In this example, by returning a `long long` instead of an `int`, the overflow defect is fixed.

```
long long left_shift(void) {

    int foo = 33;
    return 1LL << foo;
}
```

Check Information

Group: 03. Integers (INT)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

INT34-C

Introduced in R2019a

CERT C++: INT35-C

Use correct integer precisions

Description

Rule Definition

Use correct integer precisions.

Polyspace Implementation

This checker checks for **Integer precision exceeded**.

Examples

Integer precision exceeded

Issue

Integer precision exceeded occurs when an integer expression uses the integer size in an operation that exceeds the integer precision. On some architectures, the size of an integer in memory can include sign and padding bits. On these architectures, the integer size is larger than the precision which is just the number of bits that represent the value of the integer.

Risk

Using the size of an integer in an operation on the integer precision can result in integer overflow, wrap around, or unexpected results. For instance, an unsigned integer can be stored in memory in 64 bits, but uses only 48 bits to represent its value. A 56 bits left-shift operation on this integer is undefined behavior.

Assuming that the size of an integer is equal to its precision can also result in program portability issues between different architectures.

Fix

Do not use the size of an integer instead of its precision. To determine the integer precision, implement a precision computation routine or use a builtin function such as `__builtin_popcount()`.

Example - Using Size of `unsigned int` for Left Shift Operation

```
#include <limits.h>

unsigned int func(unsigned int exp)
{
    if (exp >= sizeof(unsigned int) * CHAR_BIT) {
        /* Handle error */
    }
    return 1U << exp;
}
```

In this example, the function uses a left shift operation to return the value of 2 raised to the power of `exp`. The operation shifts the bits of `1U` by `exp` positions to the left. The `if` statement ensures that

the operation does not shift the bits by a number of positions `exp` greater than the size of an `unsigned int`. However, if `unsigned int` contains padding bits, the value returned by `sizeof()` is larger than the precision of `unsigned int`. As a result, some values of `exp` might be too large, and the shift operation might be undefined behavior.

Correction — Implement Function to Compute Precision of `unsigned int`

One possible correction is to implement a function `popcount()` that computes the precision of `unsigned int` by counting the number of set bits.

```
#include <stddef.h>
#include <stdint.h>
#include <limits.h>

size_t popcount(uintmax_t);
#define PRECISION(umax_value) popcount(umax_value)


unsigned int func(unsigned int exp)
{
    if (exp >= PRECISION(UINT_MAX)) {
        /* Handle error */
    }
    return 1 << exp;
}




size_t popcount(uintmax_t num)
{
    size_t precision = 0;
    while (num != 0) {
        if (num % 2 == 1) {
            precision++;
        }
        num >>= 1;
    }
    return precision;
}
```

Check Information

Group: 03. Integers (INT)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

INT35-C

Introduced in R2019a

CERT C++: INT36-C

Converting a pointer to integer or integer to pointer

Description

Rule Definition

Converting a pointer to integer or integer to pointer.

Polyspace Implementation

This checker checks for **Unsafe conversion between pointer and integer**.

Examples

Unsafe conversion between pointer and integer

Issue

Unsafe conversion between pointer and integer checks for pointer to integer and integer to pointers conversions. If you convert between a pointer, `intptr_t`, or `uintprt_t` and an integer type, such as `enum`, `ptrdiff_t`, or `pid_t`, Polyspace raises a defect.

Risk

The mapping between pointers and integers is not always consistent with the addressing structure of the environment.

Converting from pointers to integers can create:

- Truncated or out of range integer values.
- Invalid integer types.

Converting from integers to pointers can create:

- Misaligned pointers or misaligned objects.
- Invalid pointer addresses.

Fix

Where possible, avoid pointer-to-integer or integer-to-pointer conversions. If you want to convert a `void` pointer to an integer, so that you do not change the value, use types:

- C99 — `intptr_t` or `uintptr_t`
- C90 — `size_t` or `ssize_t`

Example - Integer to Pointer Conversions

```
unsigned int *badintptrcast(void)
{
    unsigned int *ptr0 = (unsigned int *)0xdeadbeef;
    char *ptr1 = (char *)0xdeadbeef;
```

```
    return (unsigned int *)(ptr0 - (unsigned int *)ptr1);
}
```

In this example, there are three conversions, two unsafe conversions and one safe conversion. The first conversion of `0xdeadbeef` to `unsigned int*` causes alignment issues for the pointer. The second conversion of `0xdeadbeef` to `char *` is safe because there are no alignment issues for `char`. The third conversion in the return casts `ptrdiff_t` to a pointer. This pointer might or might not point to an invalid address.

Correction — Use `intptr_t`

One possible correction is to use `intptr_t` types to store the pointer address `0xdeadbeef`. Also, you can change the second pointer to an integer offset so that there is no longer a conversion from `ptrdiff_t` to a pointer.

```
#include <stdint.h>

unsigned int *badintptrcast(void)
{
    intptr_t iptr0 = (intptr_t)0xdeadbeef;
    int offset = 0;
    return (unsigned int *)(iptr0 - offset);
}
```

Check Information

Group: 03. Integers (INT)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

INT36-C

Introduced in R2019a

CERT C++: ARR30-C

Do not form or use out-of-bounds pointers or array subscripts

Description

Rule Definition

Do not form or use out-of-bounds pointers or array subscripts.

Polyspace Implementation

This checker checks for these issues:

- **Array access out of bounds**.
- **Pointer access out of bounds**.
- **Array access with tainted index**.
- **Use of tainted pointer**.
- **Pointer dereference with tainted offset**.

Examples

Array access out of bounds

Issue

Array access out of bounds occurs when an array index falls outside the range `[0...array_size-1]` during array access.

Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back

using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Array Access Out of Bounds Error

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

Correction — Keep Array Index Within Array Bounds

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

Pointer access out of bounds

Issue

Pointer access out of bounds occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

Array access with tainted index

Issue

Array access with tainted index detects reading or writing to an array by using a tainted index that has not been validated.

Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

Example - Use Index to Return Buffer Value

```
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    return tab[num];
}
```

In this example, the index `num` accesses the array `tab`. The function does not check to see if `num` is inside the range of `tab`.

Correction — Check Range Before Use

One possible correction is to check that `num` is in range before using it.

```
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    if (num >= 0 && num < SIZE100) {
           return tab[num];
    } else {
        return -9999;
    }
}
```

Use of tainted pointer

Issue

Use of tainted pointer defect is raised when:

- Tainted NULL pointer — the pointer is not validated against NULL.
- Tainted size pointer — the size of the memory zone that a pointer points to is not validated.

Note On a single pointer, your code can have instances of **Use of tainted pointer**, **Pointer dereference with tainted offset**, and **Tainted NULL or non-null-terminated string**. Bug Finder raises only the first tainted pointer defect that it finds.

Risk

An attacker can give your program a pointer that points to unexpected memory locations. If the pointer is dereferenced to write, the attacker can:

- Modify the state variables of a critical program.
- Cause your program to crash.
- Execute unwanted code.

If the pointer is dereferenced to read, the attacker can:

- Read sensitive data.
- Cause your program to crash.
- Modify a program variable to an unexpected value.

Fix

Avoid use of pointers from external sources.

Alternatively, if you trust the external source, sanitize the pointer before dereference. In a separate sanitization function:

- Check that the pointer is not NULL.
- Check the size of the memory location (if possible). This second check validates whether the size of the data the pointer points to matches the size your program expects.

The defect still appears in the body of the sanitization function. However, if you use a sanitization function, instead of several occurrences, the defect appears only once. You can justify the defect and

hide it in later reviews by using code annotations. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Function That Dereferences an External Pointer

```
void taintedptr(int* p, int i) {
    *p = i;
}
```

In this example, the pointer `*p` is passed as an argument, and the value is changed. The pointer can be null or point to unknown memory, which can be vulnerable.

Correction — Avoid Use of External Pointers

One possible correction is to avoid pointers from external sources.

```
int *taintedptr(int i) {
    /* Use heap memory allocated in the application */
    int *p = (int *)malloc(sizeof (int));
    if (p != NULL) { /* Check for success */
        *p = i;
    }
return p;
}
```

Correction — Check Pointer

Another possible correction is to sanitize the pointer before using it. This example uses a second function to check if the pointer is null and can be dereferenced.

```
#include <stdlib.h>

int* sanitize_ptr(int* p) {
    int* res = NULL;
    if (p && *p) { /* Tainted pointer detected here, used as "firewall" */
        /* Pointer is not null and dereference ok */
        res = p;
    }
    return res;
}
void taintedptr(int* p, int i) {
    p = sanitize_ptr(p);
    if (p) {
        *p = i;
    }
}
```

Pointer dereference with tainted offset

Issue

Pointer dereference with tainted offset detects pointer dereferencing, either reading or writing, using an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

Fix

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

Example - Dereference Pointer Array

```
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(int i) {
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[i];
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `i`. The value of `i` could be outside the pointer range, causing an out-of-range error.

Correction — Check Index Before Dereference

One possible correction is to validate the value of the index. If the index is inside the valid range, continue with the pointer dereferencing.

```
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);
```

```
int taintedptroffset(int i) {
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (i>0 && i<SIZE10) {
            c = pint[i];
        }
        free(pint);
    }
    return c;
}
```

Check Information

Group: 04. Containers (CTR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

"Check for Coding Standard Violations"

External Websites

ARR30-C

Introduced in R2019a

CERT C++: ARR37-C

Do not add or subtract an integer to a pointer to a non-array object

Description

Rule Definition

Do not add or subtract an integer to a pointer to a non-array object.

Polyspace Implementation

This checker checks for **Invalid assumptions about memory organization**.

Examples

Invalid assumptions about memory organization

Issue

Invalid assumptions about memory organization occurs when you compute the address of a variable in the stack by adding or subtracting from the address of another non-array variable.

Risk

When you compute the address of a variable in the stack by adding or subtracting from the address of another variable, you assume a certain memory organization. If your assumption is incorrect, accessing the computed address can be invalid.

Fix

Do not perform an access that relies on assumptions about memory organization.

Example - Reliance on Memory Organization

```
void func(void) {
    int var1 = 0x00000011, var2;
    *(&var1 + 1) = 0;
}
```

In this example, the programmer relies on the assumption that `&var1 + 1` provides the address of `var2`. Therefore, an **Invalid assumptions about memory organization** appears on the `+` operation. In addition, a **Pointer access out of bounds** error also appears on the dereference.

Correction — Do Not Rely on Memory Organization

One possible correction is not perform direct computation on addresses to access separately declared variables.

Check Information

Group: 04. Containers (CTR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

ARR37-C

Introduced in R2019a

CERT C++: ARR38-C

Guarantee that library functions do not form invalid pointers

Description

Rule Definition

Guarantee that library functions do not form invalid pointers.

Polyspace Implementation

This checker checks for these issues:

- **Mismatch between data length and size**.
- **Invalid use of standard library memory routine**.
- **Possible misuse of sizeof**.
- **Buffer overflow from incorrect string format specifier**.
- **Invalid use of standard library string routine**.
- **Destination buffer overflow in string manipulation**.
- **Destination buffer underflow in string manipulation**.

Examples

Mismatch between data length and size

Issue

Mismatch between data length and size looks for memory copying functions such as `memcpy`, `memset`, or `memmove`. If you do not control the length argument and data buffer argument properly, Bug Finder raises a defect.

Risk

If an attacker can manipulate the data buffer or length argument, the attacker can cause buffer overflow by making the actual data size smaller than the length.

This mismatch in length allows the attacker to copy memory past the data buffer to a new location. If the extra memory contains sensitive information, the attacker can now access that data.

This defect is similar to the SSL Heartbleed bug.

Fix

When copying or manipulating memory, compute the length argument directly from the data so that the sizes match.

Example - Copy Buffer of Data

```
#include <stdlib.h>
#include <string.h>
```

```
typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    memcpy(&(beta.data[num]), os->data + 2, length);

    return(1);
}
```

This function copies the buffer `alpha` into a buffer `beta`. However, the `length` variable is not related to `data+2`.

Correction — Check Buffer Length

One possible correction is to check the length of your buffer against the maximum value minus 2. This check ensures that you have enough space to copy the data to the `beta` structure.

```
#include <stdlib.h>
#include <string.h>

typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    if (length<(os->max -2)) {
        memcpy(&(beta.data[num]), os->data + 2, length);
    }

    return(1);

}
```

Invalid use of standard library memory routine

Issue

Invalid use of standard library memory routine occurs when a memory library function is called with invalid arguments. For instance, the `memcpy` function copies to an array that cannot accommodate the number of bytes copied.

Risk

Use of a memory library function with invalid arguments can result in issues such as buffer overflow.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Invalid Use of Standard Library Memory Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string `str2` is 5, but six characters of string `str1` are copied into `str2` using the `memcpy` function.

Correction — Call Function with Valid Arguments

One possible correction is to adjust the size of `str2` so that it accommodates the characters copied with the `memcpy` function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];

  printf("Enter string:\n");
```

```
  scanf("%s",str1);

  memcpy(str2,str1,6);
  return str2;
 }
```

Possible misuse of sizeof

Issue

Possible misuse of sizeof occurs when Polyspace Bug Finder detects possibly unintended results from the use of `sizeof` operator. For instance:

- You use the `sizeof` operator on an array parameter name, expecting the array size. However, the array parameter name by itself is a pointer. The `sizeof` operator returns the size of that pointer.
- You use the `sizeof` operator on an array element, expecting the array size. However, the operator returns the size of the array element.
- The size argument of certain functions such as `strncmp` or `wcsncpy` is incorrect because you used the `sizeof` operator earlier with possibly incorrect expectations. For instance:
  - In a function call `strncmp(string1, string2, num)`, `num` is obtained from an incorrect use of the `sizeof` operator on a pointer.
  - In a function call `wcsncpy(destination, source, num)`, `num` is the not the number of wide characters but a size in bytes obtained by using the `sizeof` operator. For instance, you use `wcsncpy(destination, source, sizeof(destination) - 1)` instead of `wcsncpy(destination, source, (sizeof(desintation)/sizeof(wchar_t)) - 1)`.

Risk

Incorrect use of the `sizeof` operator can cause the following issues:

- If you expect the `sizeof` operator to return array size and use the return value to constrain a loop, the number of loop runs are smaller than what you expect.
- If you use the return value of `sizeof` operator to allocate a buffer, the buffer size is smaller than what you require. Insufficient buffer can lead to resultant weaknesses such as buffer overflows.
- If you use the return value of `sizeof` operator incorrectly in a function call, the function does not behave as you expect.

Fix

Possible fixes are:

- Do not use the `sizeof` operator on an array parameter name or array element to determine array size.

  The best practice is to pass the array size as a separate function parameter and use that parameter in the function body.
- Use the `sizeof` operator carefully to determine the number argument of functions such as `strncmp` or `wcsncpy`. For instance, for wide string functions such as `wcsncpy`, use the number of wide characters as argument instead of the number of bytes.

Example - `sizeof` Used Incorrectly to Determine Array Size

```
#define MAX_SIZE 1024
```

```
void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < sizeof(a)/sizeof(int); i++)    {
        a[i] = i + 1;
    }
}
```

In this example, `sizeof(a)` returns the size of the pointer `a` and not the array size.

Correction — Determine Array Size in Another Way

One possible correction is to use another means to determine the array size.

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < MAX_SIZE; i++)    {
        a[i] = i + 1;
    }
}
```

Buffer overflow from incorrect string format specifier

Issue

Buffer overflow from incorrect string format specifier occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

Risk

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

Fix

Use a format specifier that is compatible with the memory buffer size.

Example - Memory Buffer Overflow

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf);
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

Correction — Use Smaller Precision in Format Specifier

One possible correction is to use a smaller precision in the format specifier.

```
#include <stdio.h>

void func (char *str[]) {
```

```
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

Invalid use of standard library string routine

Issue

Invalid use of standard library string routine occurs when a string library function is called with invalid arguments.

Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Invalid Use of Standard Library String Routine Error

```
 #include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>
```

```
char* Copy_String(void)
{
 char *res;
 /*Fix: gbuffer has equal or larger size than text */
 char gbuffer[20],text[20]="ABCDEFGHIJKL";

 res=strcpy(gbuffer,text);

 return(res);
}
```

Destination buffer overflow in string manipulation

Issue

Destination buffer overflow in string manipulation occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

Example - Buffer Overflow in `sprintf` Use

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

Destination buffer underflow in string manipulation

Issue

Destination buffer underflow in string manipulation occurs when certain string manipulation functions write to their destination buffer argument at a negative offset from the beginning of the buffer.

For instance, for the function `sprintf(char* buffer, const char* format)`, you obtain the `buffer` from an operation `buffer = (char*)arr; ... buffer += offset;`. `arr` is an array and `offset` is a negative value.

Risk

Buffer underflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer underflow also introduces the risk of code injection.

Fix

If the destination buffer argument results from pointer arithmetic, see if you are decrementing a pointer. Fix the pointer decrement by modifying either the original value before decrement or the decrement value.

Example - Buffer Underflow in `sprintf` Use

```
#include <stdio.h>
#define offset -2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

In this example, `&buffer[offset]` is at a negative offset from the memory allocated to `buffer`.

Correction — Change Pointer Decrementer

One possible correction is to change the value of `offset`.

```
#include <stdio.h>
#define offset 2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

Check Information

Group: 04. Containers (CTR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

ARR38-C

Introduced in R2019a

CERT C++: ARR39-C

Do not add or subtract a scaled integer to a pointer

Description

Rule Definition

Do not add or subtract a scaled integer to a pointer.

Polyspace Implementation

This checker checks for **Incorrect pointer scaling**.

Examples

Incorrect pointer scaling

Issue

Incorrect pointer scaling occurs when Polyspace Bug Finder considers that you are ignoring the implicit scaling in pointer arithmetic.

For instance, the defect can occur in the following situations.

Situation	Risk	Possible Fix
You use the `sizeof` operator in arithmetic operations on a pointer.	The `sizeof` operator returns the size of a data type in number of bytes. Pointer arithmetic is already implicitly scaled by the size of the data type of the pointed variable. Therefore, the use of `sizeof` in pointer arithmetic produces unintended results.	Do not use `sizeof` operator in pointer arithmetic.
You perform arithmetic operations on a pointer, and then apply a cast.	Pointer arithmetic is implicitly scaled. If you do not consider this implicit scaling, casting the result of a pointer arithmetic produces unintended results.	Apply the cast before the pointer arithmetic.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Use of `sizeof` Operator

```
void func(void) {
    int arr[5] = {1,2,3,4,5};
    int *ptr = arr;

    int value_in_position_2 = *(ptr + 2*(sizeof(int)));
}
```

In this example, the operation `2*(sizeof(int))` returns twice the size of an `int` variable in bytes. However, because pointer arithmetic is implicitly scaled, the number of bytes by which `ptr` is offset is `2*(sizeof(int))*(sizeof(int))`.

In this example, the incorrect scaling shifts `ptr` outside the bounds of the array. Therefore, a **Pointer access out of bounds** error appears on the * operation.

Correction — Remove `sizeof` Operator

One possible correction is to remove the `sizeof` operator.

```
void func(void) {
    int arr[5] = {1,2,3,4,5};
    int *ptr = arr;

    int value_in_position_2 = *(ptr + 2);
}
```

Example - Cast Following Pointer Arithmetic

```
int func(void) {
    int x = 0;
    char r = *(char *)(&x + 1);
    return r;
}
```

In this example, the operation `&x + 1` offsets `&x` by `sizeof(int)`. Following the operation, the resulting pointer points outside the allowed buffer. When you dereference the pointer, a **Pointer access out of bounds** error appears on the * operation.

Correction — Apply Cast Before Pointer Arithmetic

If you want to access the second byte of `x`, first cast `&x` to a `char*` pointer and then perform the pointer arithmetic. The resulting pointer is offset by `sizeof(char)` bytes and still points within the allowed buffer, whose size is `sizeof(int)` bytes.

```
int func(void) {
    int x = 0;
    char r = *((char *)(&x )+ 1);
    return r;
}
```

Check Information

Group: 04. Containers (CTR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

ARR39-C

Introduced in R2019a

CERT C++: CTR50-CPP

Guarantee that container indices and iterators are within the valid range

Description

Rule Definition

Guarantee that container indices and iterators are within the valid range.

Polyspace Implementation

This checker checks for these issues:

- **Array access out of bounds**.
- **Array access with tainted index**.
- **Pointer dereference with tainted offset**.

Examples

Array access out of bounds

Issue

Array access out of bounds occurs when an array index falls outside the range `[0...array_size-1]` during array access.

Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Array Access Out of Bounds Error

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

Correction — Keep Array Index Within Array Bounds

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

Array access with tainted index

Issue

Array access with tainted index detects reading or writing to an array by using a tainted index that has not been validated.

Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

Example - Use Index to Return Buffer Value

```
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    return tab[num];
}
```

In this example, the index `num` accesses the array `tab`. The function does not check to see if `num` is inside the range of `tab`.

Correction — Check Range Before Use

One possible correction is to check that `num` is in range before using it.

```
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    if (num >= 0 && num < SIZE100) {
           return tab[num];
    } else {
        return -9999;
    }
}
```

Pointer dereference with tainted offset

Issue

Pointer dereference with tainted offset detects pointer dereferencing, either reading or writing, using an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

Fix

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

Example - Dereference Pointer Array

```
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(int i) {
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[i];
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `i`. The value of `i` could be outside the pointer range, causing an out-of-range error.

Correction — Check Index Before Dereference

One possible correction is to validate the value of the index. If the index is inside the valid range, continue with the pointer dereferencing.

```
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(int i) {
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
```

```
        read_pint(pint);
        if (i>0 && i<SIZE10) {
            c = pint[i];
        }
        free(pint);
    }
    return c;
}
```

Check Information

Group: 04. Containers (CTR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

CTR50-CPP

Introduced in R2019a

CERT C++: STR30-C

Do not attempt to modify string literals

Description

Rule Definition

Do not attempt to modify string literals.

Polyspace Implementation

This checker checks for **Writing to const qualified object**.

Examples

Writing to const qualified object

Issue

Writing to `const` qualified object occurs when you do one of the following:

- Use a `const`-qualified object as the destination of an assignment.
- Pass a `const`-qualified object to a function that modifies the argument.

For instance, the defect can occur in the following situations:

- You pass a `const`-qualified object as first argument of one of the following functions:
  - `mkstemp`
  - `mkostemp`
  - `mkostemps`
  - `mkdtemp`
- You pass a `const`-qualified object as the destination argument of one of the following functions:
  - `strcpy`
  - `strncpy`
  - `strcat`
  - `memset`
- You perform a write operation on a `const`-qualified object.

Risk

The risk depends upon the modifications made to the `const`-qualified object.

Situation	Risk
Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on.	These functions replace the last six characters of their first argument with a string. Therefore, they expect a modifiable `char` array as their first argument.
Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on.	These functions modify their destination argument. Therefore, they expect a modifiable `char` array as their destination argument.
Writing to the object	The `const` qualifier implies an agreement that the value of the object will not be modified. By writing to a `const`-qualified object, you break the agreement. The result of the operation is undefined.

Fix

The fix depends on the modification made to the `const`-qualified object.

Situation	Fix
Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on.	Pass a non-`const` object as first argument of the function.
Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on.	Pass a non-`const` object as destination argument of the function.
Writing to the object	Perform the write operation on a non-`const` object.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Writing to `const`-Qualified Object

```
#include <string.h>

const char* buffer = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

In this example, because `buffer` is `const`-qualified, `strchr(buffer,'X')` returns a `const`-qualified `char*` pointer. When this `char*` pointer is used as the destination argument of `strcpy`, a **Writing to const qualified object** error appears.

Correction — Copy `const`-Qualified Object to Non-`const` Object

One possible correction is to assign the constant string to a non-`const` object and use the non-`const` object as destination argument of `strchr`.

```
#include <string.h>
```

```
char buffer[] = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

Check Information

Group: 05. Characters and Strings (STR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

STR30-C

Introduced in R2019a

CERT C++: STR31-C

Guarantee that storage for strings has sufficient space for character data and the null terminator

Description

Rule Definition

Guarantee that storage for strings has sufficient space for character data and the null terminator.

Polyspace Implementation

This checker checks for these issues:

- **Use of dangerous standard function**.
- **Missing null in string array**.
- **Buffer overflow from incorrect string format specifier**.
- **Destination buffer overflow in string manipulation**.

Examples

Use of dangerous standard function

Issue

The **Use of dangerous standard function** check highlights uses of functions that are inherently dangerous or potentially dangerous given certain circumstances. The following table lists possibly dangerous functions, the risks of using each function, and what function to use instead.

Dangerous Function	Risk Level	Safer Function
`gets`	Inherently dangerous — You cannot control the length of input from the console.	`fgets`
`cin`	Inherently dangerous — You cannot control the length of input from the console.	Avoid or prefaces calls to `cin` with `cin.width`.
`strcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`strncpy`
`stpcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`stpncpy`
`lstrcpy` or `StrCpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy`

Dangerous Function	Risk Level	Safer Function
`strcat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`strncat`, `strlcat`, or `strcat_s`
`lstrcat` or `StrCat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`StringCbCat`, `StringCchCat`, `strncay`, `strcat_s`, or `strlcat`
`wcpcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`wcpncpy`
`wcscat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`wcsncat`, `wcslcat`, or `wcncat_s`
`wcscpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`wcsncpy`
`sprintf`	Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur.	`snprintf`
`vsprintf`	Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur.	`vsnprintf`

Risk

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Using `sprintf`

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;
```

```
    if (sprintf(dst, "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

Correction — Use `snprintf` with Buffer Size

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

Missing null in string array

Issue

Missing null in string array occurs when a string does not have enough space to terminate with a null character `'\0'`.

This defect applies only for projects in C.

Risk

A buffer overflow can occur if you copy a string to an array without assuming the implicit null terminator.

Fix

If you initialize a character array with a literal, avoid specifying the array bounds.

```
char three[]  = "THREE";
```

The compiler automatically allocates space for a null terminator. In the preceding example, the compiler allocates sufficient space for five characters and a null terminator.

If the issue occurs after initialization, you might have to increase the size of the array by one to account for the null terminator.

In certain circumstances, you might want to initialize the character array with a sequence of characters instead of a string. In this situation, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Array size is too small

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[5] = "THREE";
}
```

The character array `three` has a size of 5 and 5 characters `'T'`, `'H'`, `'R'`, `'E'`, and `'E'`. There is no room for the null character at the end because `three` is only five bytes large.

Correction — Increase Array Size

One possible correction is to change the array size to allow for the five characters plus a null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[6] = "THREE";
}
```

Correction — Change Initialization Method

One possible correction is to initialize the string by leaving the array size blank. This initialization method allocates enough memory for the five characters and a terminating-null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[]  = "THREE";
}
```

Buffer overflow from incorrect string format specifier

Issue

Buffer overflow from incorrect string format specifier occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

Risk

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

Fix

Use a format specifier that is compatible with the memory buffer size.

Example - Memory Buffer Overflow

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf);
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

Correction — Use Smaller Precision in Format Specifier

One possible correction is to use a smaller precision in the format specifier.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

Destination buffer overflow in string manipulation

Issue

Destination buffer overflow in string manipulation occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

Example - Buffer Overflow in `sprintf` Use

```
#include <stdio.h>

void func(void) {
```

```
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

Check Information

Group: 05. Characters and Strings (STR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

STR31-C

Introduced in R2019a

CERT C++: STR32-C

Do not pass a non-null-terminated character sequence to a library function that expects a string

Description

Rule Definition

Do not pass a non-null-terminated character sequence to a library function that expects a string.

Polyspace Implementation

This checker checks for these issues:

- **Invalid use of standard library string routine**.
- **Tainted NULL or non-null-terminated string**.

Examples

Invalid use of standard library string routine

Issue

Invalid use of standard library string routine occurs when a string library function is called with invalid arguments.

Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Invalid Use of Standard Library String Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_String(void)
```

```
{
 char *res;
 char gbuffer[5],text[20]="ABCDEFGHIJKL";

 res=strcpy(gbuffer,text);
 /* Error: Size of text is less than gbuffer */

 return(res);
}
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

Tainted NULL or non-null-terminated string

Issue

Tainted NULL or non-null-terminated string looks for strings from unsecure sources that are being used in string manipulation routines that implicitly dereference the string buffer. For example, `strcpy` or `sprintf`.

Tainted NULL or non-null-terminated string raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

Note If you reference a string using the form `ptr[i]`, `*ptr`, or pointer arithmetic, Bug Finder raises a **Use of tainted pointer** defect instead. The **Tainted NULL or non-null-terminated string** defect is raised only when the pointer is used as a string.

Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

Example - Getting String from Input Argument

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

Correction — Validate the Data

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

int sanitize_str(char* s) {
  int res = 0;
  if (s && (strlen(s) > 0)) { // TAINTED_STRING only flagged here
    // - string is not null
    // - string has a positive and limited size
    // - TAINTED_STRING on strlen used as a firewall
    res = 1;
  }
```

```
  return res;
}

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
      strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  if (sanitize_str(userstr))
    strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}
```

Correction — Validate the Data

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

Check Information

Group: 05. Characters and Strings (STR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

STR32-C

Introduced in R2019a

CERT C++: STR34-C

Cast characters to unsigned char before converting to larger integer sizes

Description

Rule Definition

Cast characters to unsigned char before converting to larger integer sizes.

Polyspace Implementation

This checker checks for **Misuse of sign-extended character value**.

Examples

Misuse of sign-extended character value

Issue

Misuse of sign-extended character value occurs when you convert a signed or plain `char` data type to a wider integer data type with sign extension. You then use the resulting sign-extended value as array index, for comparison with EOF or as argument to a character-handling function.

Risk

Comparison with EOF: Suppose, your compiler implements the plain `char` type as signed. In this implementation, the character with the decimal form of 255 (–1 in two's complement form) is stored as a signed value. When you convert a `char` variable to the wider data type `int` for instance, the sign bit is preserved (sign extension). This sign extension results in the character with the decimal form 255 being converted to the integer –1, which cannot be distinguished from EOF.

Use as array index: By similar reasoning, you cannot use sign-extended plain `char` variables as array index. If the sign bit is preserved, the conversion from `char` to `int` can result in negative integers. You must use positive integer values for array index.

Argument to character-handling function: By similar reasoning, you cannot use sign-extended plain `char` variables as arguments to character-handling functions declared in `ctype.h`, for instance, `isalpha()` or `isdigit()`. According to the C11 standard (Section 7.4), if you supply an integer argument that cannot be represented as `unsigned char` or EOF, the resulting behavior is undefined.

Fix

Before conversion to a wider integer data type, cast the signed or plain `char` value explicitly to `unsigned char`.

Example - Sign-Extended Character Value Compared with EOF

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];
```

```
static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = *buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

In this example, the function `parser` can traverse a string input `buf`. If a character in the string has the decimal form 255, when converted to the `int` variable `c`, its value becomes –1, which is indistinguishable from `EOF`. The later comparison with `EOF` can lead to a false positive.

Correction — Cast to `unsigned char` Before Conversion

One possible correction is to cast the plain `char` value to `unsigned char` before conversion to the wider `int` type.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];

static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = (unsigned char)*buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

Check Information

Group: 05. Characters and Strings (STR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics
“Check for Coding Standard Violations”

External Websites
STR34-C

Introduced in R2019a

CERT C++: STR37-C

Arguments to character-handling functions must be representable as an unsigned char

Description

Rule Definition

Arguments to character-handling functions must be representable as an unsigned char.

Polyspace Implementation

This checker checks for **Invalid use of standard library integer routine**.

Examples

Invalid use of standard library integer routine

Issue

Invalid use of standard library integer routine occurs when you use invalid arguments with an integer function from the standard library. This defect picks up:

- Character Conversion

  ```
  toupper, tolower
  ```
- Character Checks

  ```
  isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct,
  isspace, isupper, isxdigit
  ```
- Integer Division

  ```
  div, ldiv
  ```
- Absolute Values

  ```
  abs, labs
  ```

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Absolute Value of Large Negative

```
#include <limits.h>
#include <stdlib.h>
```

```
int absoluteValue(void) {

    int neg = INT_MIN;
    return abs(neg);
}
```

The input value to `abs` is `INT_MIN`. The absolute value of `INT_MIN` is `INT_MAX+1`. This number cannot be represented by the type `int`.

Correction — Change Input Argument

One possible correction is to change the input value to fit returned data type. In this example, change the input value to `INT_MIN+1`.

```
#include <limits.h>
#include <stdlib.h>

int absoluteValue(void) {

    int neg = INT_MIN+1;
    return abs(neg);
}
```

Check Information

Group: 05. Characters and Strings (STR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

STR37-C

Introduced in R2019a

CERT C++: STR38-C

Do not confuse narrow and wide character strings and functions

Description

Rule Definition

Do not confuse narrow and wide character strings and functions.

Polyspace Implementation

This checker checks for **Misuse of narrow or wide character string**.

Examples

Misuse of narrow or wide character string

Issue

Misuse of narrow or wide character string occurs when you pass a narrow character string to a wide string function, or a wide character string to a narrow string function.

Misuse of narrow or wide character string raises no defect on operating systems where narrow and wide character strings have the same size.

Risk

Using a narrow character string with a wide string function, or vice versa, can result in unexpected or undefined behavior.

If you pass a wide character string to a narrow string function, you can encounter these issues:

- Data truncation. If the string contains null bytes, a copy operation using `strncpy()` can terminate early.
- Incorrect string length. `strlen()` returns the number of characters of a string up to the first null byte. A wide string can have additional characters after its first null byte.

If you pass a narrow character string to a wide string function, you can encounter this issue:

- Buffer overflow. In a copy operation using `wcsncpy()`, the destination string might have insufficient memory to store the result of the copy.

Fix

Use the narrow string functions with narrow character strings. Use the wide string functions with wide character strings.

Example - Passing Wide Character Strings to `strncpy()`

```
#include <string.h>
#include <wchar.h>

void func(void)
```

```
{
    wchar_t wide_str1[]  = L"0123456789";
    wchar_t wide_str2[] =  L"0000000000";
    strncpy(wide_str2, wide_str1, 10);
}
```

In this example, `strncpy()` copies 10 wide characters from `wide_strt1` to `wide_str2`. If `wide_str1` contains null bytes, the copy operation can end prematurely and truncate the wide character string.

Correction — Use `wcsncpy()` to Copy Wide Character Strings

One possible correction is to use `wcsncpy()` to copy `wide_str1` to `wide_str2`.

```
#include <string.h>
#include <wchar.h>

void func(void)
{
    wchar_t wide_str1[]  = L"0123456789";
    wchar_t wide_str2[] =  L"0000000000";
    wcsncpy(wide_str2, wide_str1, 10);
}
```

Check Information

Group: 05. Characters and Strings (STR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

STR38-C

Introduced in R2019a

CERT C++: STR50-CPP

Guarantee that storage for strings has sufficient space for character data and the null terminator

Description

Rule Definition

Guarantee that storage for strings has sufficient space for character data and the null terminator.

Polyspace Implementation

This checker checks for these issues:

- **Use of dangerous standard function**.
- **Missing null in string array**.
- **Buffer overflow from incorrect string format specifier**.
- **Destination buffer overflow in string manipulation**.

Examples

Use of dangerous standard function

Issue

The **Use of dangerous standard function** check highlights uses of functions that are inherently dangerous or potentially dangerous given certain circumstances. The following table lists possibly dangerous functions, the risks of using each function, and what function to use instead.

Dangerous Function	Risk Level	Safer Function
`gets`	Inherently dangerous — You cannot control the length of input from the console.	`fgets`
`cin`	Inherently dangerous — You cannot control the length of input from the console.	Avoid or prefaces calls to `cin` with `cin.width`.
`strcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`strncpy`
`stpcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`stpncpy`
`lstrcpy` or `StrCpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy`

Dangerous Function	Risk Level	Safer Function
`strcat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`strncat`, `strlcat`, or `strcat_s`
`lstrcat` or `StrCat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`StringCbCat`, `StringCchCat`, `strncay`, `strcat_s`, or `strlcat`
`wcpcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`wcpncpy`
`wcscat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`wcsncat`, `wcslcat`, or `wcncat_s`
`wcscpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`wcsncpy`
`sprintf`	Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur.	`snprintf`
`vsprintf`	Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur.	`vsnprintf`

Risk

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Using `sprintf`

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;
```

```
    if (sprintf(dst, "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

Correction — Use `snprintf` with Buffer Size

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

Missing null in string array

Issue

Missing null in string array occurs when a string does not have enough space to terminate with a null character `'\0'`.

This defect applies only for projects in C.

Risk

A buffer overflow can occur if you copy a string to an array without assuming the implicit null terminator.

Fix

If you initialize a character array with a literal, avoid specifying the array bounds.

```
char three[]  = "THREE";
```

The compiler automatically allocates space for a null terminator. In the preceding example, the compiler allocates sufficient space for five characters and a null terminator.

If the issue occurs after initialization, you might have to increase the size of the array by one to account for the null terminator.

In certain circumstances, you might want to initialize the character array with a sequence of characters instead of a string. In this situation, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Array size is too small

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[5] = "THREE";
}
```

The character array `three` has a size of 5 and 5 characters `'T'`, `'H'`, `'R'`, `'E'`, and `'E'`. There is no room for the null character at the end because `three` is only five bytes large.

Correction — Increase Array Size

One possible correction is to change the array size to allow for the five characters plus a null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[6] = "THREE";
}
```

Correction — Change Initialization Method

One possible correction is to initialize the string by leaving the array size blank. This initialization method allocates enough memory for the five characters and a terminating-null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[]  = "THREE";
}
```

Buffer overflow from incorrect string format specifier

Issue

Buffer overflow from incorrect string format specifier occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

Risk

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

Fix

Use a format specifier that is compatible with the memory buffer size.

Example - Memory Buffer Overflow

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf);
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

Correction — Use Smaller Precision in Format Specifier

One possible correction is to use a smaller precision in the format specifier.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

Destination buffer overflow in string manipulation

Issue

Destination buffer overflow in string manipulation occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

Example - Buffer Overflow in `sprintf` Use

```
#include <stdio.h>

void func(void) {
```

```
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

Check Information

Group: 05. Characters and Strings (STR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

STR50-CPP

Introduced in R2019a

CERT C++: STR53-CPP

Range check element access

Description

Rule Definition

Range check element access.

Polyspace Implementation

This checker checks for these issues:

- **Array access out of bounds**.
- **Array access with tainted index**.
- **Pointer dereference with tainted offset**.

Examples

Array access out of bounds

Issue

Array access out of bounds occurs when an array index falls outside the range `[0...array_size-1]` during array access.

Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Array Access Out of Bounds Error

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

Correction — Keep Array Index Within Array Bounds

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

Array access with tainted index

Issue

Array access with tainted index detects reading or writing to an array by using a tainted index that has not been validated.

Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

Example - Use Index to Return Buffer Value

```
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    return tab[num];
}
```

In this example, the index `num` accesses the array `tab`. The function does not check to see if `num` is inside the range of `tab`.

Correction — Check Range Before Use

One possible correction is to check that `num` is in range before using it.

```
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    if (num >= 0 && num < SIZE100) {
           return tab[num];
    } else {
        return -9999;
    }
}
```

Pointer dereference with tainted offset

Issue

Pointer dereference with tainted offset detects pointer dereferencing, either reading or writing, using an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

Fix

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

Example - Dereference Pointer Array

```
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(int i) {
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[i];
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `i`. The value of `i` could be outside the pointer range, causing an out-of-range error.

Correction — Check Index Before Dereference

One possible correction is to validate the value of the index. If the index is inside the valid range, continue with the pointer dereferencing.

```
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(int i) {
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
```

```
        read_pint(pint);
        if (i>0 && i<SIZE10) {
            c = pint[i];
        }
        free(pint);
    }
    return c;
}
```

Check Information

Group: 05. Characters and Strings (STR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

STR53-CPP

Introduced in R2019a

CERT C++: MEM30-C

Do not access freed memory

Description

Rule Definition

Do not access freed memory.

Polyspace Implementation

This checker checks for **Use of previously freed pointer**.

Examples

Use of previously freed pointer

Issue

Use of previously freed pointer occurs when you access a block of memory after freeing the block using the `free` function.

Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to access this block of memory can result in unpredictable behavior or even a segmentation fault.

Fix

The fix depends on the root cause of the defect. See if you intended to free the memory later or allocate another memory block to the pointer before access.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before dereferencing pointers, check them for NULL values and handle the error. In this way, you are protected against accessing a freed block.

Example - Use of Previously Freed Pointer Error

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   {
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);

    j = *pi + shift;
    /* Defect: Reading a freed pointer */
```

```
    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, dereferencing`pi` after the `free` statement is not valid.

Correction — Free Pointer After Use

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
#include <stdlib.h>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

Check Information

Group: 06. Memory Management (MEM)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MEM30-C

Introduced in R2019a

CERT C++: MEM31-C

Free dynamically allocated memory when no longer needed

Description

Rule Definition

Free dynamically allocated memory when no longer needed.

Polyspace Implementation

This checker checks for **Memory leak**.

Examples

Memory leak

Issue

Memory leak occurs when you do not free a block of memory allocated through `malloc`, `calloc`, `realloc`, or `new`. If the memory is allocated in a function, the defect does not occur if:

- Within the function, you free the memory using `free` or `delete`.
- The function returns the pointer assigned by `malloc`, `calloc`, `realloc`, or `new`.
- The function stores the pointer in a global variable or in a parameter.

Risk

Dynamic memory allocation functions such as `malloc` allocate memory on the heap. If you do not release the memory after use, you reduce the amount of memory available for another allocation. On embedded systems with limited memory, you might end up exhausting available heap memory even during program execution.

Fix

Determine the scope where the dynamically allocated memory is accessed. Free the memory block at the end of this scope.

To free a block of memory, use the `free` function on the pointer that was used during memory allocation. For instance:

```
ptr = (int*)malloc(sizeof(int));
...
free(ptr);
```

It is a good practice to allocate and free memory in the same module at the same level of abstraction. For instance, in this example, `func` allocates and frees memory at the same level but `func2` does not.

```
void func() {
  ptr = (int*)malloc(sizeof(int));
  {
    ...
```

```
  }
  free(ptr);
}

void func2() {
  {
   ptr = (int*)malloc(sizeof(int));
   ...
  }
  free(ptr);
}
```

See CERT-C Rule MEM00-C.

Example - Dynamic Memory Not Released Before End of Function

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }


    *pi = 42;
    /* Defect: pi is not freed */
}
```

In this example, `pi` is dynamically allocated by `malloc`. The function `assign_memory` does not free the memory, nor does it return `pi`.

Correction — Free Memory

One possible correction is to free the memory referenced by `pi` using the `free` function. The `free` function must be called before the function `assign_memory` terminates

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }
    *pi = 42;

    /* Fix: Free the pointer pi*/
    free(pi);
}
```

Correction — Return Pointer from Dynamic Allocation

Another possible correction is to return the pointer `pi`. Returning `pi` allows the function calling `assign_memory` to free the memory block using `pi`.

```
#include<stdlib.h>
#include<stdio.h>

int* assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
            printf("Memory allocation failed");
            return(pi);
        }
    *pi = 42;

    /* Fix: Return the pointer pi*/
    return(pi);
}
```

Example - Memory Leak with New/Delete

```
#define NULL '\0'

void initialize_arr1(void)
{
    int *p_scalar = new int(5);
}

void initialize_arr2(void)
{
    int *p_array = new int[5];
}
```

In this example, the functions create two variables, `p_scalar` and `p_array`, using the `new` keyword. However, the functions end without cleaning up the memory for these pointers. Because the functions used `new` to create these variables, you must clean up their memory by calling `delete` at the end of each function.

Correction — Add Delete

To correct this error, add a `delete` statement for every `new` initialization. If you used brackets `[]` to instantiate a variable, you must call delete with brackets as well.

```
#define NULL '\0'

void initialize_arrs(void)
{
    int *p_scalar = new int(5);
    int *p_array = new int[5];

    delete p_scalar;
    p_scalar = NULL;
```

```
    delete[] p_array;
    p_scalar = NULL;
}
```

Check Information

Group: 06. Memory Management (MEM)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MEM31-C

Introduced in R2019a

CERT C++: MEM34-C

Only free memory allocated dynamically

Description

Rule Definition

Only free memory allocated dynamically.

Polyspace Implementation

This checker checks for **Invalid free of pointer**.

Examples

Invalid free of pointer

Issue

Invalid free of pointer occurs when a block of memory released using the `free` function was not previously allocated using `malloc`, `calloc`, or `realloc`.

Risk

The `free` function releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.

The issue can highlight coding errors. For instance, you perhaps wanted to use the `free` function or a previous `malloc` function on a different pointer.

Fix

In most cases, you can fix the issue by removing the `free` statement. If the pointer is not allocated memory from the heap with `malloc` or `calloc`, you do not need to free the pointer. You can simply reuse the pointer as required.

If the issue highlights a coding error such as use of `free` or `malloc` on the wrong pointer, correct the error.

If the issue occurs because you use the `free` function to free memory allocated with the `new` operator, replace the `free` function with the `delete` operator.

Example - Invalid Free of Pointer Error

```
#include <stdlib.h>

void Assign_Ones(void)
{
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;

  free(p);
```

```
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer `p` is deallocated using the `free` function. However, `p` points to a memory location that was not dynamically allocated.

Correction — Remove Pointer Deallocation

If the number of elements of the array `p` is known at compile time, one possible correction is to remove the deallocation of the pointer `p`.

```
#include <stdlib.h>

void Assign_Ones(void)
 {
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;
  /* Fix: Remove deallocation of p */
 }
```

Correction — Introduce Pointer Allocation

If the number of elements of the array `p` is not known at compile time, one possible correction is to dynamically allocate memory to the array `p`.

```
#include <stdlib.h>

void Assign_Ones(int num)
{
  int *p;
  /* Fix: Allocate memory dynamically to p */
  p=(int*) calloc(10,sizeof(int));
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
}
```

Check Information

Group: 06. Memory Management (MEM)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MEM34-C

Introduced in R2019a

CERT C++: MEM35-C

Allocate sufficient memory for an object

Description

Rule Definition

Allocate sufficient memory for an object.

Polyspace Implementation

This checker checks for these issues:

- **Pointer access out of bounds**.
- **Memory allocation with tainted size**.

Examples

Pointer access out of bounds

Issue

Pointer access out of bounds occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

Memory allocation with tainted size

Issue

Memory allocation with tainted size checks memory allocation functions, such as `calloc` or `malloc`, for size arguments from unsecured sources.

Risk

Uncontrolled memory allocation can cause your program to request too much system memory. This consequence can lead to a crash due to an out-of-memory condition, or assigning too many resources.

Fix

Before allocating memory, check the value of your arguments to check that they do not exceed the bounds.

Example - Allocate Memory Using Input Argument

```
#include "stdlib.h"

int* bug_taintedmemoryallocsize(size_t size) {
    int* p = (int*)malloc(size);
    return p;
}
```

In this example, `malloc` allocates `size` amount of memory for the pointer `p`. `size` is an outside variable, so could be any size value. If the size is larger than the amount of memory you have available, your program could crash.

Correction — Check Size of Memory to be Allocated

One possible correction is to check the size of the memory that you want to allocate before performing the `malloc` operation. This example checks to see if the size is positive and less than the maximum size.

```
#include "stdlib.h"

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int* corrected_taintedmemoryallocsize(int size) {
    int* p = NULL;
    if (size>0 && size<SIZE128) {          /* Fix: Check entry range before use */
        p = (int*)malloc((unsigned int)size);
    }
    return p;
}
```

Check Information

Group: 06. Memory Management (MEM)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MEM35-C

Introduced in R2019a

CERT C++: MEM36-C

Do not modify the alignment of objects by calling realloc()

Description

Rule Definition

Do not modify the alignment of objects by calling realloc().

Polyspace Implementation

This checker checks for **Alignment changed after memory reallocation**.

Examples

Alignment changed after memory reallocation

Issue

Alignment changed after memory reallocation occurs when you use `realloc()` to modify the size of objects with strict memory alignment requirements.

Risk

The pointer returned by `realloc()` can be suitably assigned to objects with less strict alignment requirements. A misaligned memory allocation can lead to buffer underflow or overflow, an illegally dereferenced pointer, or access to arbitrary memory locations. In processors that support misaligned memory, the allocation impacts the performance of the system.

Fix

To reallocate memory:

1. Resize the memory block.
   - In Windows, use `_aligned_realloc()` with the alignment argument used in `_aligned_malloc()` to allocate the original memory block.
   - In UNIX/Linux, use the same function with the same alignment argument used to allocate the original memory block.
2. Copy the original content to the new memory block.
3. Free the original memory block.

Note This fix has implementation-defined behavior. The implementation might not support the requested memory alignment and can have additional constraints for the size of the new memory.

Example - Memory Reallocated Without Preserving the Original Alignment

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define SIZE1024 1024

void func(void)
{
    size_t resize = SIZE1024;
    size_t alignment = 1 << 12; /* 4096 bytes alignment */
    int *ptr = NULL;
    int *ptr1;

    /* Allocate memory with 4096 bytes alignment */

    if (posix_memalign((void **)&ptr, alignment, sizeof(int)) != 0)
    {
        /* Handle error */
      }

    /*Reallocate memory without using the original alignment.
    ptr1 may not be 4096 bytes aligned. */

    ptr1 = (int *)realloc(ptr, sizeof(int) * resize);

    if (ptr1 == NULL)
    {
        /* Handle error */
    }

    /* Processing using ptr1 */

    /* Free before exit */
    free(ptr1);
}
```

In this example, the allocated memory is 4096-bytes aligned. `realloc()` then resizes the allocated memory. The new pointer `ptr1` might not be 4096-bytes aligned.

Correction — Specify the Alignment for the Reallocated Memory

When you reallocate the memory, use `posix_memalign()` and pass the alignment argument that you used to allocate the original memory.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE1024 1024

void func(void)
{
    size_t resize = SIZE1024;
    size_t alignment = 1 << 12; /* 4096 bytes alignment */
    int *ptr = NULL;

    /* Allocate memory with 4096 bytes alignment */
    if (posix_memalign((void **)&ptr, alignment, sizeof(int)) != 0)
    {
        /* Handle error */
```

```
    }

    /* Reallocate memory using the original alignment. */
    if (posix_memalign((void **)&ptr, alignment, sizeof(int) * resize) != 0)
    {
        /* Handle error */
        free(ptr);
        ptr = NULL;
    }

    /* Processing using ptr */

    /* Free before exit */
    free(ptr);
}
```

Check Information

Group: 06. Memory Management (MEM)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MEM36-C

Introduced in R2019a

CERT C++: MEM50-CPP

Do not access freed memory

Description

Rule Definition

Do not access freed memory.

Polyspace Implementation

This checker checks for these issues:

- **Pointer access out of bounds**.
- **Deallocation of previously deallocated pointer**.
- **Use of previously freed pointer**.

Examples

Pointer access out of bounds

Issue

Pointer access out of bounds occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

Deallocation of previously deallocated pointer

Issue

Deallocation of previously deallocated pointer occurs when a block of memory is freed more than once using the `free` function without an intermediate allocation.

Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to free this block of memory can result in a segmentation fault.

Fix

The fix depends on the root cause of the defect. See if you intended to allocate a memory block to the pointer between the first deallocation and the second. Otherwise, remove the second `free` statement.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before freeing pointers, check them for NULL values and handle the error. In this way, you are protected against freeing an already freed block.

Example - Deallocation of Previously Deallocated Pointer Error

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    free (pi);
    /* Defect: pi has already been freed */
}
```

The first `free` statement releases the block of memory that `pi` refers to. The second `free` statement on `pi` releases a block of memory that has been freed already.

Correction — Remove Duplicate Deallocation

One possible correction is to remove the second `free` statement.

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    /* Fix: remove second deallocation */
 }
```

Use of previously freed pointer

Issue

Use of previously freed pointer occurs when you access a block of memory after freeing the block using the `free` function.

Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to access this block of memory can result in unpredictable behavior or even a segmentation fault.

Fix

The fix depends on the root cause of the defect. See if you intended to free the memory later or allocate another memory block to the pointer before access.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before dereferencing pointers, check them for NULL values and handle the error. In this way, you are protected against accessing a freed block.

Example - Use of Previously Freed Pointer Error

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   {
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);

    j = *pi + shift;
    /* Defect: Reading a freed pointer */

    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, dereferencing`pi` after the `free` statement is not valid.

Correction — Free Pointer After Use

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
#include <stdlib.h>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

Check Information

Group: 06. Memory Management (MEM)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MEM50-CPP

Introduced in R2019a

CERT C++: MEM51-CPP

Properly deallocate dynamically allocated resources

Description

Rule Definition

Properly deallocate dynamically allocated resources.

Polyspace Implementation

This checker checks for these issues:

- **Invalid deletion of pointer**.
- **Invalid free of pointer**.
- **Deallocation of previously deallocated pointer**.

Examples

Invalid deletion of pointer

Issue

Invalid deletion of pointer occurs when:

- You release a block of memory with the `delete` operator but the memory was previously not allocated with the `new` operator.
- You release a block of memory with the `delete` operator using the single-object notation but the memory was previously allocated as an array with the `new` operator.

This defect applies only to C++ source files.

Risk

The risk depends on the cause of the issue:

- The `delete` operator releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.
- If you use the single-object notation for `delete` on a pointer that is previously allocated with the array notation for `new`, the behavior is undefined.

The issue can also highlight other coding errors. For instance, you perhaps wanted to use the `delete` operator or a previous `new` operator on a different pointer.

Fix

The fix depends on the cause of the issue:

- In most cases, you can fix the issue by removing the `delete` statement. If the pointer is not allocated memory from the heap with the `new` operator, you do not need to release the pointer with `delete`. You can simply reuse the pointer as required or let the object be destroyed at the end of its scope.

- In case of mismatched notation for `new` and `delete`, correct the mismatch. For instance, to allocate and deallocate a single object, use this notation:

  ```
  classType* ptr = new classType;
  delete ptr;
  ```

  To allocate and deallocate an array objects, use this notation:

  ```
  classType* p2 = new classType[10];
  delete[] p2;
  ```

If the issue highlights a coding error such as use of `delete` or `new` on the wrong pointer, correct the error.

Example - Deleting Static Memory

```
void assign_ones(void)
{
    int ptr[10];

    for(int i=0;i<10;i++)
        *(ptr+i)=1;

    delete[] ptr;
}
```

The pointer `ptr` is released using the `delete` operator. However, `ptr` points to a memory location that was not dynamically allocated.

Correction: Remove Pointer Deallocation

If the number of elements of the array `ptr` is known at compile time, one possible correction is to remove the deallocation of the pointer `ptr`.

```
void assign_ones(void)
{
    int ptr[10];

    for(int i=0;i<10;i++)
        *(ptr+i)=1;
}
```

Correction — Add Pointer Allocation

If the number of array elements is not known at compile time, one possible correction is to dynamically allocate memory to the array `ptr` using the `new` operator.

```
void assign_ones(int num)
{
    int *ptr = new int[num];

    for(int i=0; i < num; i++)
        *(ptr+i) = 1;

    delete[] ptr;
   }
```

Example - Mismatched `new` and `delete`

```
int main (void)
{
```

```
    int *p_scale = new int[5];

    //more code using scal

    delete p_scale;
}
```

In this example, `p_scale` is initialized to an array of size 5 using `new int[5]`. However, `p_scale` is deleted with `delete` instead of `delete[]`. The `new`-`delete` pair does not match. Do not use `delete` without the brackets when deleting arrays.

Correction — Match `delete` to `new`

One possible correction is to add brackets so the `delete` matches the `new []` declaration.

```
int main (void)
{
    int *p_scale = new int[5];

    //more code using p_scale

    delete[] p_scale;
}
```

Correction — Match `new` to `delete`

Another possible correction is to change the declaration of `p_scale`. If you meant to initialize `p_scale` as 5 itself instead of an array of size 5, you must use different syntax. For this correction, change the square brackets in the initialization to parentheses. Leave the `delete` statement as it is.

```
int main (void)
{
    int *p_scale = new int(5);

    //more code using p_scale

    delete p_scale;
}
```

Invalid free of pointer

Issue

Invalid free of pointer occurs when a block of memory released using the `free` function was not previously allocated using `malloc`, `calloc`, or `realloc`.

Risk

The `free` function releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.

The issue can highlight coding errors. For instance, you perhaps wanted to use the `free` function or a previous `malloc` function on a different pointer.

Fix

In most cases, you can fix the issue by removing the `free` statement. If the pointer is not allocated memory from the heap with `malloc` or `calloc`, you do not need to free the pointer. You can simply reuse the pointer as required.

If the issue highlights a coding error such as use of `free` or `malloc` on the wrong pointer, correct the error.

If the issue occurs because you use the `free` function to free memory allocated with the `new` operator, replace the `free` function with the `delete` operator.

Example - Invalid Free of Pointer Error

```
#include <stdlib.h>

void Assign_Ones(void)
{
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;

  free(p);
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer `p` is deallocated using the `free` function. However, `p` points to a memory location that was not dynamically allocated.

Correction — Remove Pointer Deallocation

If the number of elements of the array `p` is known at compile time, one possible correction is to remove the deallocation of the pointer `p`.

```
#include <stdlib.h>

void Assign_Ones(void)
 {
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;
  /* Fix: Remove deallocation of p */
 }
```

Correction — Introduce Pointer Allocation

If the number of elements of the array `p` is not known at compile time, one possible correction is to dynamically allocate memory to the array `p`.

```
#include <stdlib.h>

void Assign_Ones(int num)
{
  int *p;
  /* Fix: Allocate memory dynamically to p */
  p=(int*) calloc(10,sizeof(int));
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
}
```

Deallocation of previously deallocated pointer

Issue

Deallocation of previously deallocated pointer occurs when a block of memory is freed more than once using the `free` function without an intermediate allocation.

Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to free this block of memory can result in a segmentation fault.

Fix

The fix depends on the root cause of the defect. See if you intended to allocate a memory block to the pointer between the first deallocation and the second. Otherwise, remove the second `free` statement.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before freeing pointers, check them for NULL values and handle the error. In this way, you are protected against freeing an already freed block.

Example - Deallocation of Previously Deallocated Pointer Error

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    free (pi);
    /* Defect: pi has already been freed */
}
```

The first `free` statement releases the block of memory that `pi` refers to. The second `free` statement on `pi` releases a block of memory that has been freed already.

Correction — Remove Duplicate Deallocation

One possible correction is to remove the second `free` statement.

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    /* Fix: remove second deallocation */
 }
```

Check Information

Group: 06. Memory Management (MEM)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MEM51-CPP

Introduced in R2019a

CERT C++: MEM52-CPP

Detect and handle memory allocation errors

Description

Rule Definition

Detect and handle memory allocation errors.

Polyspace Implementation

This checker checks for **Unprotected dynamic memory allocation**.

Examples

Unprotected dynamic memory allocation

Issue

Unprotected dynamic memory allocation occurs when you do not check after dynamic memory allocation whether the memory allocation succeeded.

Risk

When memory is dynamically allocated using `malloc`, `calloc`, or `realloc`, it returns a value `NULL` if the requested memory is not available. If the code following the allocation accesses the memory block without checking for this `NULL` value, this access is not protected from failures.

Fix

Check the return value of `malloc`, `calloc`, or `realloc` for NULL before accessing the allocated memory location.

```
int *ptr = malloc(size * sizeof(int));

if(ptr) /* Check for NULL */
{
   /* Memory access through ptr */
}
```

Example - Unprotected dynamic memory allocation error

```
#include <stdlib.h>

void Assign_Value(void)
{
  int* p = (int*)calloc(5, sizeof(int));

  *p = 2;
  /* Defect: p is not checked for NULL value */

  free(p);
}
```

If the memory allocation fails, the function `calloc` returns `NULL` to `p`. Before accessing the memory through `p`, the code does not check whether `p` is `NULL`

Correction — Check for NULL Value

One possible correction is to check whether `p` has value `NULL` before dereference.

```
#include <stdlib.h>

void Assign_Value(void)
 {
   int* p = (int*)calloc(5, sizeof(int));

   /* Fix: Check if p is NULL */
   if(p!=NULL) *p = 2;

   free(p);
 }
```

Check Information

Group: 06. Memory Management (MEM)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MEM52-CPP

Introduced in R2019a

CERT C++: MEM57-CPP

Avoid using default operator new for over-aligned types

Description

Rule Definition

Avoid using default operator new for over-aligned types.

Polyspace Implementation

This checker checks for **Operator new not overloaded for possibly overaligned class**.

Examples

Operator new not overloaded for possibly overaligned class

Issue

Operator new not overloaded for possibly overaligned class occurs when you do not adequately overload operator `new/new[]` and you use this operator to create an object with an alignment requirement specified with `alignas`. The checker raises a defect for these versions of throwing and non-throwing operator `new/new[]`.

- `void* operator new(std::size_t size)`
- `void* operator new(std::size_t size, const std::nothrow_t&)`
- `void* operator new[](std::size_t size)`
- `void* operator new[](std::size_t size, const std::nothrow_t&)`

The use of `alignas` indicates that you do not expect the default operator `new/new[]` to satisfy the alignment requirement or the object, and that the object is possibly over aligned. A type is over aligned if you use `alignas` to make the alignment requirement of the type larger than `std::max_align_t`. For instance, `foo` is over aligned in this code snippet because its alignment requirement is 32 bytes, but `std::max_align_t` has an alignment of 16 bytes in most implementations.

```
struct alignas(32) foo {
  char elems[32];
}
```

Operator new not overloaded for possibly overaligned class raises no defect if you do not overload the operator `new/new[]` and you use version C++17 or later of the Standard. The default operator `new/new[]` in C++17 or later supports over alignment by passing the alignment requirement as an argument of type `std::align_val_t`, for instance `void* operator new(std::size_t size, std::align_val_t alignment)`.

Risk

The default operator `new/new[]` allocates storage with the alignment requirement of `std::align_val_t` at most. If you do not overload the operator when you create an object with

over aligned type, the resulting object may be misaligned. Accessing this object might cause illegal access errors or abnormal program terminations.

Fix

If you use version C++14 or earlier of the Standard, pass the alignment requirement of over aligned types to the operator `new/new[]` by overloading the operator.

Example - Allocated Memory Is Smaller Than Alignment Requirement of Type `foo`

```
#include <new>
#include <cstdlib>
#include <iostream>

struct alignas(64) foo {
    char elems[32];
};

foo*  func()
{
    foo*  bar = 0x0;
    try {
        bar =  new  foo ;
    } catch (...) { return nullptr; }
    delete bar;
}
```

In this example, structure `foo` is declared with an alignment requirement of 32 bytes. When you use the default operator `new` to create object `bar`, the allocated memory for `bar` is smaller than the alignment requirement of type `foo` and `bar` might be misaligned.

Correction — Define Overloaded Operator `new` to Handle Alignment Requirement of Type `foo`

One possible correction, if you use C11 `stdlib.h` or POSIX-C `malloc.h`, is to define an overloaded operator `new` that uses `aligned_alloc()` or `posix_memalign()` or to obtain storage with the correct alignment.

```
#include <new>
#include <cstdlib>
#include <iostream>

struct alignas(64) foo {
    char elems[32];
    static void* operator new (size_t nbytes)
    {
        if (void* p =
                ::aligned_alloc(alignof(foo), nbytes)) {
            return p;
        }
        throw std::bad_alloc();
    }
    static void operator delete(void *p) {
        free(p);
    }
};

foo*  func()
{
    foo*  bar = 0x0;
    try {
        bar =  new  foo ;
    } catch (...) { return nullptr; }
    delete bar;
}
```

Check Information

Group: Rule 06. Memory Management (MEM)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics
“Check for Coding Standard Violations”

External Websites
MEM57-CPP

Introduced in R2019b

CERT C++: FIO30-C

Exclude user input from format strings

Description

Rule Definition

Exclude user input from format strings.

Polyspace Implementation

This checker checks for **Tainted string format**.

Examples

Tainted string format

Issue

Tainted string format detects string formatting with `printf`-style functions that contain elements from unsecure sources.

Risk

If you use externally controlled elements to format a string, you can cause buffer overflow or data-representation problems. An attacker can use these string formatting elements to view the contents of a stack using `%x` or write to a stack using `%n`.

Fix

Pass a static string to format string functions. This fix ensures that an external actor cannot control the string.

Another possible fix is to allow only the expected number of arguments. If possible, use functions that do not support the vulnerable `%n` operator in format strings.

Example - Get Elements from User Input

```
#include "stdio.h"

void taintedstringformat(char* userstr) {
    printf(userstr);
}
```

This example prints the input argument `userstr`. The string is unknown. If it contains elements such as `%`, `printf` can interpret `userstr` as a string format instead of a string, causing your program to crash.

Correction — Print as String

One possible correction is to print `userstr` explicitly as a string so that there is no ambiguity.

```
#include "stdio.h"
```

```
void taintedstringformat(char* userstr) {
    printf("%.20s", userstr);
}
```

Check Information

Group: 07. Input Output (FIO)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FIO30-C

Introduced in R2019a

CERT C++: FIO32-C

Do not perform operations on devices that are only appropriate for files

Description

Rule Definition

Do not perform operations on devices that are only appropriate for files.

Polyspace Implementation

This checker checks for **Inappropriate I/O operation on device files**.

Examples

Inappropriate I/O operation on device files

Issue

Inappropriate I/O operation on device files occurs when you do not check whether a file name parameter refers to a device file before you pass it to these functions:

- `fopen()`
- `fopen_s()`
- `freopen()`
- `remove()`
- `rename()`
- `CreateFile()`
- `CreateFileA()`
- `CreateFileW()`
- `_wfopen()`
- `_wfopen_s()`

Device files are files in a file system that provide an interface to device drivers. You can use these files to interact with devices.

Inappropriate I/O operation on device files does not raise a defect when:

- You use `stat` or `lstat`-family functions to check the file name parameter before calling the previously listed functions.
- You use a string comparison function to compare the file name against a list of device file names.

Risk

Operations appropriate only for regular files but performed on device files can result in denial-of-service attacks, other security vulnerabilities, or system failures.

Fix

Before you perform an I/O operation on a file:

- Use `stat()`, `lstat()`, or an equivalent function to check whether the file name parameter refers to a regular file.
- Use a string comparison function to compare the file name against a list of device file names.

Example - Using `fopen()` Without Checking `file_name`

```
#include <stdio.h>
#include <string.h>

#define SIZE1024 1024

FILE* func()
{

    FILE* f;
    const char file_name[SIZE1024] = "./tmp/file";

    if ((f = fopen(file_name, "w")) == NULL) {
        /*handle error */
    };
    /*operate on file */
}
```

In this example, `func()` operates on the file `file_name` without checking whether it is a regular file. If `file_name` is a device file, attempts to access it can result in a system failure.

Correction — Check File with `lstat()` Before Calling `fopen()`

One possible correction is to use `lstat()` and the `S_ISREG` macro to check whether the file is a regular file. This solution contains a `TOCTOU` race condition that can allow an attacker to modify the file after you check it but before the call to fopen(). To prevent this vulnerability, ensure that `file_name` refers to a file in a secure folder.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>

#define SIZE1024 1024

FILE* func()
{

    FILE* f;
    const char file_name[SIZE1024] = "./tmp/file";
    struct stat orig_st;
    if ((lstat(file_name, &orig_st) != 0) ||
        (!S_ISREG(orig_st.st_mode))) {
        exit(0);
    }
    if ((f = fopen(file_name, "w")) == NULL) {
        /*handle error */
    };
```

```
    /*operate on file */
}
```

Check Information

Group: 07. Input Output (FIO)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FIO32-C

Introduced in R2019a

CERT C++: FIO34-C

Distinguish between characters read from a file and EOF or WEOF

Description

Rule Definition

Distinguish between characters read from a file and EOF or WEOF.

Polyspace Implementation

This checker checks for **Character value absorbed into EOF**.

Examples

Character value absorbed into EOF

Issue

Character value absorbed into EOF occurs when you perform a data type conversion that makes a valid character value indistinguishable from `EOF` (End-of-File). Bug Finder flags the defect in one of the following situations:

- *End-of-File*: You perform a data type conversion such as from `int` to `char` that converts a non-EOF character value into `EOF`.

  ```
  char ch = (char)getchar()
  ```

  You then compare the result with EOF.

  ```
  if((int)ch == EOF)
  ```

  The conversion can be explicit or implicit.
- *Wide End-of-File*: You perform a data type conversion that can convert a non-WEOF wide character value into WEOF, and then compare the result with WEOF.

Risk

The data type `char` cannot hold the value `EOF` that indicates the end of a file. Functions such as `getchar` have return type `int` to accommodate `EOF`. If you convert from `int` to `char`, the values `UCHAR_MAX` (a valid character value) and `EOF` get converted to the same value -1 and become indistinguishable from each other. When you compare the result of this conversion with `EOF`, the comparison can lead to false detection of `EOF`. This rationale also applies to wide character values and `WEOF`.

Fix

Perform the comparison with `EOF` or `WEOF` before conversion.

Example - Return Value of `getchar` Converted to `char`

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define fatal_error() abort()

char func(void)
{
    char ch;
    ch = getchar();
    if (EOF == (int)ch) {
        fatal_error();
    }
    return ch;
}
```

In this example, the return value of `getchar` is implicitly converted to `char`. If `getchar` returns `UCHAR_MAX`, it is converted to -1, which is indistinguishable from EOF. When you compare with EOF later, it can lead to a false positive.

Correction — Perform Comparison with EOF Before Conversion

One possible correction is to first perform the comparison with EOF, and then convert from `int` to `char`.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

char func(void)
{
    int i;
    i = getchar();
    if (EOF == i) {
        fatal_error();
    }
    else {
        return (char)i;
    }
}
```

Check Information

Group: 07. Input Output (FIO)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FIO34-C

Introduced in R2019a

CERT C++: FIO37-C

Do not assume that fgets() or fgetws() returns a nonempty string when successful

Description

Rule Definition

Do not assume that fgets() or fgetws() returns a nonempty string when successful.

Polyspace Implementation

This checker checks for **Use of indeterminate string**.

Examples

Use of indeterminate string

Issue

Use of indeterminate string occurs when you do not check the validity of the buffer returned from `fgets`-family functions. The checker raises a defect when such a buffer is used as:

- An argument in standard functions that print or manipulate strings or wide strings.
- A return value.
- An argument in external functions with parameter type `const char *` or `const wchar_t *`.

Risk

If an `fgets`-family function fails, the content of its output buffer is indeterminate. Use of such a buffer has undefined behavior and can result in a program that stops working or other security vulnerabilities.

Fix

Reset the output buffer of an `fgets`-family function to a known string value when the function fails.

Example - Output of `fgets()` Passed to External Function

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func(void) {
    char buf[SIZE20];

    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
```

```
        /* 'buf' may contain an indeterminate string.  */
        ;
    }
    /* 'buf passed to external function */
    display_text(buf);
}
```

In this example, the output `buf` is passed to the external function `display_text()`, but its value is not reset if `fgets()` fails.

Correction — Reset `fgets()` Output on Failure

If `fgets()` fails, reset `buf` to a known value before you pass it to an external function.

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func1(void) {
    char buf[SIZE20];
    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* value of 'buf' reset after fgets() failure. */
        buf[0] = '\0';
    }
    /* 'buf' passed to external function */
    display_text(buf);
}
```

Check Information

Group: 07. Input Output (FIO)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FIO37-C

Introduced in R2019a

CERT C++: FIO38-C

Do not copy a FILE object

Description

Rule Definition

Do not copy a FILE object.

Polyspace Implementation

This checker checks for **Misuse of a FILE object**.

Examples

Misuse of a FILE object

Issue

Misuse of a FILE object occurs when:

- You dereference a pointer to a FILE object, including indirect dereference by using `memcmp()`.
- You modify an entire FILE object or one of its components through its pointer.
- You take the address of FILE object that was not returned from a call to an `fopen`-family function. No defect is raised if a macro defines the pointer as the address of a built-in FILE object, such as `#define ptr (&__stdout)`.

Risk

In some implementations, the address of the pointer to a FILE object used to control a stream is significant. A pointer to a copy of a FILE object is interpreted differently than a pointer to the original object, and can potentially result in operations on the wrong stream. Therefore, the use of a copy of a FILE object can cause the software to stop responding, which an attacker might exploit in denial-of-service attacks.

Fix

Do not make a copy of a FILE object. Do not use the address of a FILE object that was not returned from a successful call to an `fopen`-family function.

Example - Copy of FILE Object Used in `fputs()`

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

void fatal_error(void);

int func(void)
{
```

```
    /*'stdout' dereferenced and contents
        copied to 'my_stdout'. */
    FILE my_stdout = *stdout;

    /* Address of 'my_stdout' may not point to correct stream. */
    if (fputs("Hello, World!\n", &my_stdout) == EOF)
    {
        /* Handler error */
        fatal_error();
    }
    return 0;
}
```

In this example, FILE object `stdout` is dereferenced and its contents are copied to `my_stdout`. The contents of `stdout` might not be significant. `fputs()` is then called with the address of `my_stdout` as an argument. Because no call to `fopen()` or a similar function was made, the address of `my_stdout` might not point to the correct stream.

Correction — Copy the FILE Object Pointer

Declare `my_stdout` to point to the same address as `stdout` to ensure that you write to the correct stream when you call `fputs()`.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

void fatal_error(void);

int func(void)
{
    /* 'my_stdout' and 'stdout' point to the same object. */
    FILE *my_stdout = stdout;
    if (fputs("Hello, World!\n", my_stdout) == EOF)
    {
        /* Handler error */
        fatal_error();
    }
    return 0;
}
```

Check Information

Group: 07. Input Output (FIO)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FIO38-C

Introduced in R2019a

CERT C++: FIO39-C

Do not alternately input and output from a stream without an intervening flush or positioning call

Description

Rule Definition

Do not alternately input and output from a stream without an intervening flush or positioning call.

Polyspace Implementation

This checker checks for **Alternating input and output from a stream without flush or positioning call**.

Examples

Alternating input and output from a stream without flush or positioning call

Issue

Alternating input and output from a stream without flush or positioning call occurs when:

- You do not perform a flush or function positioning call between an output operation and a following input operation on a file stream in update mode.
- You do not perform a function positioning call between an input operation and a following output operation on a file stream in update mode.

Risk

Alternating input and output operations on a stream without an intervening flush or positioning call is undefined behavior.

Fix

Call `fflush()` or a file positioning function such as `fseek()` or `fsetpos()` between output and input operations on an update stream.

Call a file positioning function between input and output operations on an update stream.

Example - Read After Write Without Intervening Flush

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";

void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;
```

```
    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Read operation after write without
    intervening flush. */
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
          (void)fclose(file);
          /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

In this example, the file `demo.txt` is opened for reading and appending. After the call to `fwrite()`, a call to `fread()` without an intervening flush operation is undefined behavior.

Correction — Call `fflush()` Before the Read Operation

After writing data to the file, before calling `fread()`, perform a flush call.

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";


void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;

    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
```

```
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Buffer flush after write and before read */
    if (fflush(file) != 0)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

Check Information

Group: 07. Input Output (FIO)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FIO39-C

Introduced in R2019a

CERT C++: FIO40-C

Reset strings on fgets() or fgetws() failure

Description

Rule Definition

Reset strings on fgets() or fgetws() failure.

Polyspace Implementation

This checker checks for **Use of indeterminate string**.

Examples

Use of indeterminate string

Issue

Use of indeterminate string occurs when you do not check the validity of the buffer returned from `fgets`-family functions. The checker raises a defect when such a buffer is used as:

- An argument in standard functions that print or manipulate strings or wide strings.
- A return value.
- An argument in external functions with parameter type `const char *` or `const wchar_t *`.

Risk

If an `fgets`-family function fails, the content of its output buffer is indeterminate. Use of such a buffer has undefined behavior and can result in a program that stops working or other security vulnerabilities.

Fix

Reset the output buffer of an `fgets`-family function to a known string value when the function fails.

Example - Output of `fgets()` Passed to External Function

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func(void) {
    char buf[SIZE20];

    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
```

```
        /* 'buf' may contain an indeterminate string.  */
        ;
    }
    /* 'buf passed to external function */
    display_text(buf);
}
```

In this example, the output `buf` is passed to the external function `display_text()`, but its value is not reset if `fgets()` fails.

Correction — Reset `fgets()` Output on Failure

If `fgets()` fails, reset `buf` to a known value before you pass it to an external function.

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func1(void) {
    char buf[SIZE20];
    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* value of 'buf' reset after fgets() failure. */
        buf[0] = '\0';
    }
    /* 'buf' passed to external function */
    display_text(buf);
}
```

Check Information

Group: 07. Input Output (FIO)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FIO40-C

Introduced in R2019a

CERT C++: FIO41-C

Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects

Description

Rule Definition

Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects.

Polyspace Implementation

This checker checks for **Stream argument with possibly unintended side effects**.

Examples

Stream argument with possibly unintended side effects

Issue

Stream argument with possibly unintended side effects occurs when you call `getc()`, `putc()`, `getwc()`, or `putwc()` with a stream argument that has side effects.

Stream argument with possibly unintended side effects considers the following as stream side effects:

- Any assignment of a variable of a stream, such as `FILE *`, or any assignment of a variable of a deeper stream type, such as an array of `FILE *`.
- Any call to a function that manipulates a stream or a deeper stream type.

The number of defects raised corresponds to the number of side effects detected. When a stream argument is evaluated multiple times in a function implemented as a macro, a defect is raised for each evaluation that has a side effect.

A defect is also raised on functions that are not implemented as macros but that can be implemented as macros on another operating system.

Risk

If the function is implemented as an unsafe macro, the stream argument can be evaluated more than once, and the stream side effect happens multiple times. For instance, a stream argument calling `fopen()` might open the same file multiple times, which is unspecified behavior.

Fix

To ensure that the side effect of a stream happens only once, use a separate statement for the stream argument.

Example - Stream Argument of `getc()` Has Side Effect `fopen()`

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#define fatal_error() abort()

const char* myfile = "my_file.log";

void func(void)
{
    int c;
    FILE* fptr;
    /* getc() has stream argument fptr with
    * 2 side effects: call to fopen(), and assignment
    * of fptr
    */
    c = getc(fptr = fopen(myfile, "r"));
    if (c == EOF) {
        /* Handle error */
        (void)fclose(fptr);
        fatal_error();
    }
    if (fclose(fptr) == EOF) {
        /* Handle error */
        fatal_error();
    }
}

void main(void)
{
    func();

}
```

In this example, `getc()` is called with stream argument `fptr`. The stream argument has two side effects: the call to `fopen()` and the assignment of `fptr`. If `getc()` is implemented as an unsafe macro, the side effects happen multiple times.

Correction — Use Separate Statement for `fopen()`

One possible correction is to use a separate statement for `fopen()`. The call to `fopen()` and the assignment of `fptr` happen in this statement so there are no side effects when you pass `fptr` to `getc()`.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

#define fatal_error() abort()


const char* myfile = "my_file.log";

void func(void)
{
    int c;
    FILE* fptr;

    /* Separate statement for fopen()
    * before call to getc()
    */
    fptr = fopen(myfile, "r");
```

```
    if (fptr == NULL) {
        /* Handle error */
        fatal_error();
    }
    c = getc(fptr);
    if (c == EOF) {
        /* Handle error */
        (void)fclose(fptr);
        fatal_error();
    }
    if (fclose(fptr) == EOF) {
        /* Handle error */
        fatal_error();
    }
}

void main(void)
{
    func();

}
```

Check Information

Group: 07. Input Output (FIO)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FIO41-C

Introduced in R2019a

CERT C++: FIO42-C

Close files when they are no longer needed

Description

Rule Definition

Close files when they are no longer needed.

Polyspace Implementation

This checker checks for **Resource leak**.

Examples

Resource leak

Issue

Resource leak occurs when you open a file stream by using a `FILE` pointer but do not close it before:

- The end of the pointer's scope.
- Assigning the pointer to another stream.

Risk

If you do not release file handles explicitly as soon as possible, a failure can occur due to exhaustion of resources.

Fix

Close a `FILE` pointer before the end of its scope, or before you assign the pointer to another stream.

Example - `FILE` Pointer Not Released Before End of Scope

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

In this example, the file pointer `fp1` is pointing to a file `data1.txt`. Before `fp1` is explicitly dissociated from the file stream of `data1.txt`, it is used to access another file `data2.txt`.

Correction — Release `FILE` Pointer

One possible correction is to explicitly dissociate `fp1` from the file stream of `data1.txt`.

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );
    fclose(fp1);

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

Check Information

Group: 07. Input Output (FIO)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FIO42-C

Introduced in R2019a

CERT C++: FIO44-C

Only use values for fsetpos() that are returned from fgetpos()

Description

Rule Definition

Only use values for fsetpos() that are returned from fgetpos().

Polyspace Implementation

This checker checks for **Invalid file position**.

Examples

Invalid file position

Issue

Invalid file position occurs when the file position argument of `fsetpos()` uses a value that is not obtained from `fgetpos()`.

Risk

The function `fgetpos(FILE *stream, fpos_t *pos)` gets the current file position of the stream. When you use any other value as the file position argument of `fsetpos(FILE *stream, const fpos_t *pos)`, you might access an unintended location in the stream.

Fix

Use the value returned from a successful call to `fgetpos()` as the file position argument of `fsetpos()`.

Example - `memset()` Sets File Position Argument

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


FILE *func(FILE *file)
{
    fpos_t offset;
    if (file == NULL)
    {
        /* Handle error */
    }
    /* Store initial position in variable 'offset' */
    (void)memset(&offset, 0, sizeof(offset));

    /* Read data from file */

    /* Return to the initial position. offset was not
    returned from a call to fgetpos()    */
```

```
    if (fsetpos(file, &offset) != 0)
    {
        /* Handle error */
    }
    return file;
}
```

In this example, `fsetpos()` uses `offset` as its file position argument. However, the value of `offset` is set by `memset()`. The preceding code might access the wrong location in the stream.

Correction — Use a File Position Returned From `fgetpos()`

Call `fgetpos()`, and if it returns successfully, use the position argument in your call to `fsetpos()`.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

FILE *func(FILE *file)
{
    fpos_t offset;
    if (file == NULL)
    {
        /* Handle error */
    }
    /* Store initial position in variable 'offset'
    using fgetpos() */
    if (fgetpos(file, &offset) != 0)
    {
        /* Handle error */
    }

    /* Read data from file */

    /* Back to the initial position */
    if (fsetpos(file, &offset) != 0)
    {
        /* Handle error */
    }
    return file;
}
```

Check Information

Group: 07. Input Output (FIO)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FIO44-C

Introduced in R2019a

CERT C++: FIO45-C

Avoid TOCTOU race conditions while accessing files

Description

Rule Definition

Avoid TOCTOU race conditions while accessing files.

Polyspace Implementation

This checker checks for **File access between time of check and use (TOCTOU)**.

Examples

File access between time of check and use (TOCTOU)

Issue

File access between time of check and use (TOCTOU) detects race condition issues between checking the existence of a file or folder, and using a file or folder.

Risk

An attacker can access and manipulate your file between your check for the file and your use of a file. Symbolic links are particularly risky because an attacker can change where your symbolic link points.

Fix

Before using a file, do not check its status. Instead, use the file and check the results afterward.

Example - Check File Before Using

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern void print_tofile(FILE* f);

void toctou(char * log_path) {
    if (access(log_path, W_OK)==0) {
        FILE* f = fopen(log_path, "w");
        if (f) {
            print_tofile(f);
            fclose(f);
        }
    }
}
```

In this example, before opening and using the file, the function checks if the file exists. However, an attacker can change the file between the first and second lines of the function.

Correction — Open Then Check

One possible correction is to open the file, and then check the existence and contents afterward.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern void print_tofile(FILE* f);

void toctou(char * log_path) {
    int fd = open(log_path, O_WRONLY);
    if (fd!=-1) {
        FILE *f = fdopen(fd, "w");
        if (f) {
            print_tofile(f);
            fclose(f);
        }
    }
}
```

Check Information

Group: 07. Input Output (FIO)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FIO45-C

Introduced in R2019a

CERT C++: FIO46-C

Do not access a closed file

Description

Rule Definition

Do not access a closed file.

Polyspace Implementation

This checker checks for **Use of previously closed resource**.

Examples

Use of previously closed resource

Issue

Use of previously closed resource occurs when a function operates on a stream that you closed earlier in your code.

Risk

The standard states that the value of a `FILE*` pointer is indeterminate after you close the stream associated with it. Operations using the `FILE*` pointer can produce unintended results.

Fix

One possible fix is to close the stream only at the end of operations. Another fix is to reopen the stream before using it again.

Example - Use of `FILE*` Pointer After Closing Stream

```
#include <stdio.h>

void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fclose(fp);
        fprintf(fp,"text");
    }
}
```

In this example, `fclose` closes the stream associated with `fp`. When you use `fprintf` on `fp` after `fclose`, the **Use of previously closed resource** defect appears.

Correction — Close Stream After All Operations

One possible correction is to reverse the order of the `fprintf` and `fclose` operations.

```
#include <stdio.h>

void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fprintf(fp,"text");
        fclose(fp);
    }
}
```

Check Information

Group: 07. Input Output (FIO)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FIO46-C

Introduced in R2019a

CERT C++: FIO47-C

Use valid format strings

Description

Rule Definition

Use valid format strings.

Polyspace Implementation

This checker checks for **Format string specifiers and arguments mismatch**.

Examples

Format string specifiers and arguments mismatch

Issue

Format string specifiers and arguments mismatch occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
  const char *message = "License not available";
  int err_number = ;-4
  printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Printing a Float

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", fst);
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

Check Information

Group: 07. Input Output (FIO)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FIO47-C

Introduced in R2019a

CERT C++: FIO50-CPP

Do not alternately input and output from a file stream without an intervening positioning call

Description

Rule Definition

Do not alternately input and output from a file stream without an intervening positioning call.

Polyspace Implementation

This checker checks for **Alternating input and output from a stream without flush or positioning call**.

Examples

Alternating input and output from a stream without flush or positioning call

Issue

Alternating input and output from a stream without flush or positioning call occurs when:

- You do not perform a flush or function positioning call between an output operation and a following input operation on a file stream in update mode.
- You do not perform a function positioning call between an input operation and a following output operation on a file stream in update mode.

Risk

Alternating input and output operations on a stream without an intervening flush or positioning call is undefined behavior.

Fix

Call `fflush()` or a file positioning function such as `fseek()` or `fsetpos()` between output and input operations on an update stream.

Call a file positioning function between input and output operations on an update stream.

Example - Read After Write Without Intervening Flush

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";

void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;
```

```
    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Read operation after write without
    intervening flush. */
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
          (void)fclose(file);
          /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

In this example, the file `demo.txt` is opened for reading and appending. After the call to `fwrite()`, a call to `fread()` without an intervening flush operation is undefined behavior.

Correction — Call `fflush()` Before the Read Operation

After writing data to the file, before calling `fread()`, perform a flush call.

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";


void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;

    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
```

```
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Buffer flush after write and before read */
    if (fflush(file) != 0)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

Check Information

Group: 07. Input Output (FIO)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FIO50-CPP

Introduced in R2019a

CERT C++: FIO51-CPP

Close files when they are no longer needed

Description

Rule Definition

Close files when they are no longer needed.

Polyspace Implementation

This checker checks for **Resource leak**.

Examples

Resource leak

Issue

Resource leak occurs when you open a file stream by using a `FILE` pointer but do not close it before:

- The end of the pointer's scope.
- Assigning the pointer to another stream.

Risk

If you do not release file handles explicitly as soon as possible, a failure can occur due to exhaustion of resources.

Fix

Close a `FILE` pointer before the end of its scope, or before you assign the pointer to another stream.

Example - `FILE` Pointer Not Released Before End of Scope

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

In this example, the file pointer `fp1` is pointing to a file `data1.txt`. Before `fp1` is explicitly dissociated from the file stream of `data1.txt`, it is used to access another file `data2.txt`.

Correction — Release `FILE` Pointer

One possible correction is to explicitly dissociate `fp1` from the file stream of `data1.txt`.

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );
    fclose(fp1);

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

Check Information

Group: 07. Input Output (FIO)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FIO51-CPP

Introduced in R2019a

CERT C++: ERR30-C

Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure

Description

Rule Definition

Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure.

Polyspace Implementation

This checker checks for these issues:

- **Misuse of errno**.
- **Errno not reset**.

Examples

Misuse of errno

Issue

Misuse of errno occurs when you check `errno` for error conditions in situations where checking `errno` does not guarantee the absence of errors. In some cases, checking `errno` can lead to false positives.

For instance, you check `errno` following calls to the functions:

- `fopen`: If you follow the ISO Standard, the function might not set `errno` on errors.
- `atof`: If you follow the ISO Standard, the function does not set `errno`.
- `signal`: The `errno` value indicates an error only if the function returns the `SIG_ERR` error indicator.

Risk

The ISO C Standard does not enforce that these functions set `errno` on errors. Whether the functions set `errno` or not is implementation-dependent.

To detect errors, if you check `errno` alone, the validity of this check also becomes implementation-dependent.

In some cases, the `errno` value indicates an error only if the function returns a specific error indicator. If you check `errno` before checking the function return value, you can see false positives.

Fix

For information on how to detect errors, see the documentation for that specific function.

Typically, the functions return an out-of-band error indicator to indicate errors. For instance:

- `fopen` returns a null pointer if an error occurs.
- `signal` returns the `SIG_ERR` error indicator and sets `errno` to a positive value. Check `errno` only after you have checked the function return value.

Example - Incorrectly Checking for `errno` After `fopen` Call

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define fatal_error() abort()

const char *temp_filename = "/tmp/demo.txt";

FILE *func()
{
    FILE *fileptr;
    errno = 0;
    fileptr = fopen(temp_filename, "w+b");
    if (errno != 0) {
        if (fileptr != NULL) {
            (void)fclose(fileptr);
        }
        /* Handle error */
        fatal_error();
    }
    return fileptr;
}
```

In this example, `errno` is the first variable that is checked after a call to `fopen`. You might expect that `fopen` changes `errno` to a nonzero value if an error occurs. If you run this code with an implementation of `fopen` that does not set `errno` on errors, you might miss an error condition. In this situation, `fopen` can return a null pointer that escapes detection.

Correction — Check Return Value of `fopen` After Call

One possible correction is to only check the return value of `fopen` for a null pointer.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define fatal_error() abort()

const char *temp_filename = "/tmp/demo.txt";

FILE *func()
{
    FILE *fileptr;
    fileptr = fopen(temp_filename, "w+b");
    if (fileptr == NULL) {
        fatal_error();
    }
    return fileptr;
}
```

Errno not reset

Issue

Errno not reset occurs when you do not reset `errno` before calling a function that sets `errno` to indicate error conditions. However, you check `errno` for those error conditions after the function call.

Risk

The `errno` is not clean and can contain values from a previous call. Checking `errno` for errors can give the false impression that an error occurred.

`errno` is set to zero at program startup but subsequently, `errno` is not reset by a C standard library function. You must explicitly set `errno` to zero when required.

Fix

Before calling a function that sets `errno` to indicate error conditions, reset `errno` to zero explicitly.

Example - `errno` Not Reset Before Call to `strtod`

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>

#define fatal_error() abort()

double func(const char *s1, const char *s2)
{
    double f1;
    f1 = strtod (s1, NULL);
    if (0 == errno) {
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
                  {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

In this example, `errno` is not reset to 0 before the first call to `strtod`. Checking `errno` for 0 later can lead to a false positive.

Correction — Reset `errno` Before Call

One possible correction is to reset `errno` to 0 before calling `strtod`.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>
```

```
#define fatal_error() abort()

double func(const char *s1, const char *s2)
{
    double f1;
    errno = 0;
    f1 = strtod (s1, NULL);
    if (0 == errno) {
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
              {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

Check Information

Group: 08. Exceptions and Error Handling (ERR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

ERR30-C

Introduced in R2019a

CERT C++: ERR32-C

Do not rely on indeterminate values of errno

Description

Rule Definition

Do not rely on indeterminate values of errno.

Polyspace Implementation

This checker checks for **Misuse of errno in a signal handler**.

Examples

Misuse of errno in a signal handler

Issue

Misuse of errno in a signal handler occurs when you call one of these functions in a signal handler:

- `signal`: You call the `signal` function in a signal handler and then read the value of `errno`.

  For instance, the signal handler function `handler` calls `signal` and then calls `perror`, which reads `errno`.

  ```
  void handler(int signum) {
    pfv old_handler = signal(signum, SIG_DFL);
    if (old_handler == SIG_ERR) {
      perror("SIGINT handler");
    }
  }
  ```

- `errno`-setting POSIX function: You call an `errno`-setting POSIX function in a signal handler but do not restore `errno` when returning from the signal handler.

  For instance, the signal handler function `handler` calls `waitpid`, which changes `errno`, but does not restore `errno` before returning.

  ```
  void handler(int signum) {
    int rc = waitpid(-1, NULL, WNOHANG);
    if (ECHILD != errno) {
    }
  }
  ```

Risk

In each case that the checker flags, you risk relying on an indeterminate value of `errno`.

- `signal`: If the call to `signal` in a signal handler fails, the value of `errno` is indeterminate (see C11 Standard, Sec. 7.14.1.1). If you rely on a specific value of `errno`, you can see unexpected results.

- `errno`-setting POSIX function: An `errno`-setting function sets `errno` on failure. If you read `errno` after a signal handler is called and the signal handler itself calls an `errno`-setting function, you can see unexpected results.

Fix

Avoid situations where you risk relying on an indeterminate value of `errno`.

- `signal`: After calling the `signal` function in a signal handler, do not read `errno` or use a function that reads `errno`.
- `errno`-setting POSIX function: Before calling an `errno`-setting function in a signal handler, save `errno` to a temporary variable. Restore `errno` from this variable before returning from the signal handler.

Example - Reading `errno` After `signal` Call in Signal Handler

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

#define fatal_error() abort()

void handler(int signum) {
    if (signal(signum, SIG_DFL) == SIG_ERR) {
        perror("SIGINT handler");
    }
}

int func(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
        fatal_error();
    }
    /* Program code */
    if (raise(SIGINT) != 0) {
        /* Handle error */
        fatal_error();
    }
    return 0;
}
```

In this example, the function `handler` is called to handle the `SIGINT` signal. In the body of `handler`, the `signal` function is called. Following this call, the value of `errno` is indeterminate. The checker raises a defect when the `perror` function is called because `perror` relies on the value of `errno`.

Correction — Avoid Reading `errno` After `signal` Call

One possible correction is to not read `errno` after calling the `signal` function in a signal handler. The corrected code here calls the `abort` function via the `fatal_error` macro instead of the `perror` function.

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

#define fatal_error() abort()
```

```
void handler(int signum) {
    if (signal(signum, SIG_DFL) == SIG_ERR) {
        fatal_error();
    }
}

int func(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
        fatal_error();
    }
    /* Program code */
    if (raise(SIGINT) != 0) {
        /* Handle error */
        fatal_error();
    }
    return 0;
}
```

Check Information

Group: 08. Exceptions and Error Handling (ERR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

ERR32-C

Introduced in R2019a

CERT C++: ERR33-C

Detect and handle standard library errors

Description

Rule Definition

Detect and handle standard library errors.

Polyspace Implementation

This checker checks for these issues:

- **Errno not checked**.
- **Returned value of a sensitive function not checked**.
- **Unprotected dynamic memory allocation**.

Examples

Errno not checked

Issue

Errno not checked occurs when you call a function that sets `errno` to indicate error conditions, but do not check `errno` after the call. For these functions, checking `errno` is the only reliable way to determine if an error occurred.

Functions that set `errno` on errors include:

- `fgetwc`, `strtol`, and `wcstol`.

  For a comprehensive list of functions, see documentation about errno.
- POSIX `errno`-setting functions such as `encrypt` and `setkey`.

Risk

To see if the function call completed without errors, check `errno` for error values.

The return values of these `errno`-setting functions do not indicate errors. The return value can be one of the following:

- `void`
- Even if an error occurs, the return value can be the same as the value from a successful call. Such return values are called in-band error indicators.

You can determine if an error occurred only by checking `errno`.

For instance, `strtol` converts a string to a long integer and returns the integer. If the result of conversion overflows, the function returns `LONG_MAX` and sets `errno` to `ERANGE`. However, the function can also return `LONG_MAX` from a successful conversion. Only by checking `errno` can you distinguish between an error and a successful conversion.

Fix

Before calling the function, set `errno` to zero.

After the function call, to see if an error occurred, compare `errno` to zero. Alternatively, compare `errno` to known error indicator values. For instance, `strtol` sets `errno` to `ERANGE` to indicate errors.

The error message in the Polyspace result shows the error indicator value that you can compare to.

Example - `errno` Not Checked After Call to `strtol`

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    long val = strtol(str, &endptr, base);
    printf("Return value of strtol() = %ld\n", val);
}
```

You are using the return value of `strtol` without checking `errno`.

Correction — Check `errno` After Call

Before calling `strtol`, set `errno` to zero . After a call to `strtol`, check the return value for `LONG_MIN` or `LONG_MAX` and `errno` for `ERANGE`.

```
#include<stdlib.h>
#include<stdio.h>
#include<errno.h>
#include<limits.h>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    errno = 0;
    long val = strtol(str, &endptr, base);
    if((val == LONG_MIN || val == LONG_MAX) && errno == ERANGE) {
         printf("strtol error");
         exit(EXIT_FAILURE);
    }
    printf("Return value of strtol() = %ld\n", val);
}
```

Returned value of a sensitive function not checked

Issue

Returned value of a sensitive function not checked occurs when you call sensitive standard functions, but you:

- Ignore the return value.
- Use an output or a return value without testing the validity of the return value.

For this defect, two type of functions are considered: ***sensitive*** and ***critical sensitive***.

A ***sensitive*** function is a standard function that can encounter:

- Exhausted system resources (for example, when allocating resources)
- Changed privileges or permissions
- Tainted sources when reading, writing, or converting data from external sources
- Unsupported features despite an existing API

A ***critical sensitive*** function is a sensitive function that performs one of these critical or vulnerable tasks:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

Risk

If you do not check the return value of functions that perform sensitive or critical sensitive tasks, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

Fix

Before continuing with the program, test the return value of *critical sensitive* functions.

For *sensitive functions*, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

Example - Sensitive Function Return Ignored

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    pthread_attr_init(&attr);
}
```

This example shows a call to the sensitive function `pthread_attr_init`. The return value of `pthread_attr_init` is ignored, causing a defect.

Correction — Cast Function to (void)

One possible correction is to cast the function to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of the sensitive function.

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    (void)pthread_attr_init(&attr);
}
```

Correction — Test Return Value

One possible correction is to test the return value of `pthread_attr_init` to check for errors.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

void initialize() {
    pthread_attr_t attr;
    int result;

    result = pthread_attr_init(&attr);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

Example - Critical Function Return Ignored

```
#include <pthread.h>
extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0));
    pthread_join(thread_id,  &res);
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), Polyspace does not ignore this *Return value of a sensitive function not checked* defect. The other critical function, `pthread_join`, returns value that is ignored implicitly. `pthread_join` uses the return value of `pthread_create`, which was not checked.

Correction — Test the Return Value of Critical Functions

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

Unprotected dynamic memory allocation

Issue

Unprotected dynamic memory allocation occurs when you do not check after dynamic memory allocation whether the memory allocation succeeded.

Risk

When memory is dynamically allocated using `malloc`, `calloc`, or `realloc`, it returns a value `NULL` if the requested memory is not available. If the code following the allocation accesses the memory block without checking for this `NULL` value, this access is not protected from failures.

Fix

Check the return value of `malloc`, `calloc`, or `realloc` for NULL before accessing the allocated memory location.

```
int *ptr = malloc(size * sizeof(int));

if(ptr) /* Check for NULL */
{
   /* Memory access through ptr */
}
```

Example - Unprotected dynamic memory allocation error

```
#include <stdlib.h>

void Assign_Value(void)
{
  int* p = (int*)calloc(5, sizeof(int));
```

```
  *p = 2;
  /* Defect: p is not checked for NULL value */

  free(p);
}
```

If the memory allocation fails, the function `calloc` returns `NULL` to `p`. Before accessing the memory through `p`, the code does not check whether `p` is `NULL`

Correction — Check for NULL Value

One possible correction is to check whether `p` has value `NULL` before dereference.

```
#include <stdlib.h>

void Assign_Value(void)
 {
   int* p = (int*)calloc(5, sizeof(int));

   /* Fix: Check if p is NULL */
   if(p!=NULL) *p = 2;

   free(p);
 }
```

Check Information

Group: 08. Exceptions and Error Handling (ERR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

ERR33-C

Introduced in R2019a

CERT C++: ERR34-C

Detect errors when converting a string to a number

Description

Rule Definition

Detect errors when converting a string to a number.

Polyspace Implementation

This checker checks for **Unsafe conversion from string to numerical value**.

Examples

Unsafe conversion from string to numerical value

Issue

Unsafe conversion from string to numerical value detects conversions from strings to integer or floating-point values. If your conversion method does not include robust error handling, a defect is raised.

Risk

Converting a string to numerical value can cause data loss or misinterpretation. Without validation of the conversion or error handling, your program continues with invalid values.

Fix

- Add additional checks to validate the numerical value.
- Use a more robust string-to-numeric conversion function such as `strtol`, `strtoll`, `strtoul`, or `strtoull`.

Example - Conversion With `atoi`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int demo_check_string_not_empty(char *s)
{
    if (s != NULL)
        return strlen(s) > 0; /* check string null-terminated and not empty */
    else
        return 0;
}

int unsafestrtonumeric(char* argv1)
{
    int s = 0;
    if (demo_check_string_not_empty(argv1))
    {
```

```
        s = atoi(argv1);
    }
    return s;
}
```

In this example, `argv1` is converted to an integer with `atoi`. `atoi` does not provide errors for an invalid integer string. The conversion can fail unexpectedly.

Correction — Use `strtol` instead

One possible correction is to use `strtol` to validate the input string and the converted integer.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <errno.h>

static int demo_check_string_not_empty(char *s)
{
    if (s != NULL)
        return strlen(s) > 0; /* check string null-terminated and not empty */
    else
        return 0;
}

int unsafestrtonumeric(char *argv1)
{
    char *c_str = argv1;
    char *end;
    long sl;

    if (demo_check_string_not_empty(c_str))
    {
        errno = 0; /* set errno for error check */
        sl = strtol(c_str, &end, 10);
        if (end == c_str)
        {
            (void)fprintf(stderr, "%s: not a decimal number\n", c_str);
        }
        else if ('\0' != *end)
        {
            (void)fprintf(stderr, "%s: extra characters: %s\n", c_str, end);
        }
        else if ((LONG_MIN == sl || LONG_MAX == sl) && ERANGE == errno)
        {
            (void)fprintf(stderr, "%s out of range of type long\n", c_str);
        }
        else if (sl > INT_MAX)
        {
            (void)fprintf(stderr, "%ld greater than INT_MAX\n", sl);
        }
        else if (sl < INT_MIN)
        {
            (void)fprintf(stderr, "%ld less than INT_MIN\n", sl);
        }
        else
        {
```

```
            return (int)sl;
        }
    }
    return 0;
}
```

Check Information

Group: 08. Exceptions and Error Handling (ERR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

ERR34-C

Introduced in R2019a

CERT C++: ERR50-CPP

Do not abruptly terminate the program

Description

Rule Definition

Do not abruptly terminate the program.

Polyspace Implementation

This checker checks for **Implicit call to terminate() function**.

Examples

Implicit call to terminate() function

Issue

The checker flags these situations when the terminate() function can be called implicitly:

- An exception escapes uncaught. For instance:
  - Before an exception is caught, it escapes through another function that throws an uncaught exception. For instance, a catch statement or exception handler invokes a copy constructor that throws an uncaught exception.
  - A throw expression with no operand rethrows an uncaught exception.
- A class destructor throws an exception.

Check Information

Group: 08. Exceptions and Error Handling (ERR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics
“Check for Coding Standard Violations”

External Websites
ERR50-CPP

Introduced in R2019a

CERT C++: ERR51-CPP

Handle all exceptions

Description

Rule Definition

Handle all exceptions.

Polyspace Implementation

This checker checks for **Unhandled exception not caught**.

Examples

Unhandled exception not caught

Issue

The checker shows a violation if there is no `try`/`catch` in the `main` function or the `catch` does not handle all exceptions (with ellipsis `...`). The rule is not checked if a `main` function does not exist.

The checker does not determine if an exception of an unhandled type actually propagates to `main`.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Check Information

Group: 08. Exceptions and Error Handling (ERR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

ERR51-CPP

Introduced in R2019a

CERT C++: ERR52-CPP

Do not use setjmp() or longjmp()

Description

Rule Definition

Do not use setjmp() or longjmp().

Polyspace Implementation

This checker checks for **Use of setjmp/longjmp**.

Examples

Use of setjmp/longjmp

Issue

Use of setjmp/longjmp occurs when you use a combination of `setjmp` and `longjmp` or `sigsetjmp` and `siglongjmp` to deviate from normal control flow and perform non-local jumps in your code.

Risk

Using `setjmp` and `longjmp`, or `sigsetjmp` and `siglongjmp` has the following risks:

- Nonlocal jumps are vulnerable to attacks that exploit common errors such as buffer overflows. Attackers can redirect the control flow and potentially execute arbitrary code.
- Resources such as dynamically allocated memory and open files might not be closed, causing resource leaks.
- If you use `setjmp` and `longjmp` in combination with a signal handler, unexpected control flow can occur. POSIX does not specify whether `setjmp` saves the signal mask.
- Using `setjmp` and `longjmp` or `sigsetjmp` and `siglongjmp` makes your program difficult to understand and maintain.

Fix

Perform nonlocal jumps in your code using `setjmp/longjmp` or `sigsetjmp/siglongjmp` only in contexts where such jumps can be performed securely. Alternatively, use POSIX threads if possible.

In C++, to simulate throwing and catching exceptions, use standard idioms such as `throw` expressions and `catch` statements.

Example - Use of `setjmp` and `longjmp`

```
#include <setjmp.h>
#include <signal.h>

extern int update(int);
extern void print_int(int);

static jmp_buf env;
```

```
void sighandler(int signum) {
    longjmp(env, signum);
}
void func_main(int i) {
    signal(SIGINT, sighandler);
    if (setjmp(env)==0) {
        while(1) {
            /* Main loop of program, iterates until SIGINT signal catch */
            i = update(i);
        }
    } else {
        /* Managing longjmp return */
        i = -update(i);
    }

    print_int(i);
    return;
}
```

In this example, the initial return value of `setjmp` is 0. The `update` function is called in an infinite `while` loop until the user interrupts it through a signal.

In the signal handling function, the `longjmp` statement causes a jump back to `main` and the return value of `setjmp` is now 1. Therefore, the `else` branch is executed.

Correction — Use Alternative to `setjmp` and `longjmp`

To emulate the same behavior more securely, use a `volatile` global variable instead of a combination of `setjmp` and `longjmp`.

```
#include <setjmp.h>
#include <signal.h>

extern int update(int);
extern void print_int(int);

volatile sig_atomic_t eflag = 0;

void sighandler(int signum) {
     eflag = signum;                   /* Fix: using global variable */
}

void func_main(int i) {
      /* Fix: Better design to avoid use of setjmp/longjmp */
    signal(SIGINT, sighandler);
    while(!eflag) {                    /* Fix: using global variable */
        /* Main loop of program, iterates until eflag is changed */
        i = update(i);
    }

    print_int(i);
    return;
}
```

Check Information

Group: 08. Exceptions and Error Handling (ERR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

ERR52-CPP

Introduced in R2019a

CERT C++: ERR53-CPP

Do not reference base classes or class data members in a constructor or destructor function-try-block handler

Description

Rule Definition

Do not reference base classes or class data members in a constructor or destructor function-try-block handler.

Polyspace Implementation

This checker checks for **Constructor or destructor function-try-block handler references base classes or class data members**.

Examples

Constructor or destructor function-try-block handler references base classes or class data members

Issue

The issue occurs when handlers of a function-try-block implementation of a class constructor or destructor references non-static members from this class or its bases.

Check Information

Group: 08. Exceptions and Error Handling (ERR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

ERR53-CPP

Introduced in R2019a

CERT C++: ERR54-CPP

Catch handlers should order their parameter types from most derived to least derived

Description

Rule Definition

Catch handlers should order their parameter types from most derived to least derived.

Polyspace Implementation

This checker checks for these issues:

- **Exception handlers not ordered from most-derived to base class**.
- **Incorrect order of ellipsis handler**.

Examples

Exception handlers not ordered from most-derived to base class

Issue

The issue occurs when you provide multiple handlers in a single try-catch statement or function-try-block for a derived class and some or all of its bases, and the handlers are not ordered from most-derived to base class.

Incorrect order of ellipsis handler

Issue

The issue occurs when you provide multiple handlers in a single try-catch statement or function-try-block, and the ellipsis (catch-all) handler does not occur last.

Check Information

Group: 08. Exceptions and Error Handling (ERR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics
“Check for Coding Standard Violations”

External Websites
ERR54-CPP

Introduced in R2019a

CERT C++: ERR61-CPP

Catch exceptions by lvalue reference

Description

Rule Definition

Catch exceptions by lvalue reference.

Polyspace Implementation

This checker checks for **Exception object initialized by copy in catch statement**.

Examples

Exception object initialized by copy in `catch` statement

Issue

The issue occurs when a `catch` statement

```
catch (exceptionType customExc) {
  ...
}
```

initializes the exception object `customExc` by copy.

Risk

If *`exceptionType`* has a nontrivial copy constructor or if the exception thrown belongs to a class derived from *`exceptionType`*, the copying can produce object slicing or undefined behavior.

Fix

Catch the exception by reference or `const` reference.

```
catch (exceptionType &customExc) {
  ...
}
```

Example - Derived Class Exception Caught by Value

```
#include <exception>
#include <string>
#include <typeinfo>
#include <iostream>

// Class declarations
class BaseExc {
public:
    explicit BaseExc();
    virtual ~BaseExc() {};
protected:
    BaseExc(const std::string& type);
```

```
private:
    std::string _id;
};

class IOExc: public BaseExc {
public:
    explicit IOExc();
};

//Class method declarations
BaseExc::BaseExc():_id(typeid(this).name()) {
}
BaseExc::BaseExc(const std::string& type): _id(type) {
}
IOExc::IOExc(): BaseExc(typeid(this).name()) {
}

int input(void);

int main(void) {
    int rnd = input();
    try {
        if (rnd==0) {
            throw IOExc();
        } else {
            throw BaseExc();
        }
    }


    catch(BaseExc exc) {
        std::cout << "Intercept BaseExc" << std::endl;
    }
    return 0;
}
```

In this example, the `catch` statement takes a `BaseExc` object by value. Catching exceptions by value causes copying of the object. The copying can cause:

- Undefined behavior of the exception if it fails.
- Object slicing if an exception of the derived class `IOExc` is caught.

Correction — Catch Exceptions by Reference

One possible correction is to catch exceptions by reference.

```
#include <exception>
#include <string>
#include <typeinfo>
#include <iostream>

// Class declarations
class BaseExc {
public:
    explicit BaseExc();
    virtual ~BaseExc() {};
protected:
    BaseExc(const std::string& type);
```

```
private:
    std::string _id;
};

class IOExc: public BaseExc {
public:
    explicit IOExc();
};

//Class method declarations
BaseExc::BaseExc():_id(typeid(this).name()) {
}
BaseExc::BaseExc(const std::string& type): _id(type) {
}
IOExc::IOExc(): BaseExc(typeid(this).name()) {
}

int input(void);

int main(void) {
    int rnd = input();
    try {
        if (rnd==0) {
            throw IOExc();
        } else {
            throw BaseExc();
        }
    }



    catch(BaseExc& exc) {
        std::cout << "Intercept BaseExc" << std::endl;
    }
    return 0;
}
```

Check Information

Group: 08. Exceptions and Error Handling (ERR)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

ERR61-CPP

Introduced in R2019b

CERT C++: OOP51-CPP

Do not slice derived objects

Description

Rule Definition

Do not slice derived objects.

Polyspace Implementation

This checker checks for **Object slicing**.

Examples

Object slicing

Issue

Object slicing occurs when you pass a derived class object by value to a function, but the function expects a base class object as parameter.

Risk

If you pass a derived class object *by value* to a function, you expect the derived class copy constructor to be called. If the function expects a base class object as parameter:

1. The base class copy constructor is called.
2. In the function body, the parameter is considered as a base class object.

In C++, `virtual` methods of a class are resolved at run time according to the actual type of the object. Because of object slicing, an incorrect implementation of a `virtual` method can be called. For instance, the base class contains a `virtual` method and the derived class contains an implementation of that method. When you call the `virtual` method from the function body, the base class method is called, even though you pass a derived class object to the function.

Fix

One possible fix is to pass the object by reference or pointer. Passing by reference or pointer does not cause invocation of copy constructors. If you do not want the object to be modified, use a `const` qualifier with your function parameter.

Another possible fix is to overload the function with another function that accepts the derived class object as parameter.

Example - Function Call Causing Object Slicing

```
#include <iostream>

class Base {
public:
    explicit Base(int b) {
        _b = b;
```

```
    }
    virtual ~Base() {}
    virtual int update() const;
protected:
    int _b;
};


class Derived: public Base {
public:
    explicit Derived(int b):Base(b) {}
    int update() const;
};

//Class methods definition

int Base::update() const {
    return (_b + 1);
}

int Derived::update() const {
    return (_b -1);
}


//Other function definitions
void funcPassByValue(const Base bObj) {
    std::cout << "Updated _b=" << bObj.update() << std::endl;
}

int main() {
    Derived dObj(0);
    funcPassByValue(dObj);       //Function call slices object
    return 0;
 }
```

In this example, the call `funcPassByValue(dObj)` results in the output `Updated _b=1` instead of the expected `Updated _b=-1`. Because `funcPassByValue` expects a `Base` object parameter, it calls the `Base` class copy constructor.

Therefore, even though you pass the `Derived` object `dObj`, the function `funcPassByValue` treats its parameter `b` as a `Base` object. It calls `Base::update()` instead of `Derived::update()`.

Correction — Pass Object by Reference or Pointer

One possible correction is to pass the `Derived` object `dObj` by reference or by pointer. In the following, corrected example, `funcPassByReference` and `funcPassByPointer` have the same objective as `funcPassByValue` in the preceding example. However, `funcPassByReference` expects a reference to a `Base` object and `funcPassByPointer` expects a pointer to a `Base` object.

Passing the `Derived` object `d` by a pointer or by reference does not slice the object. The calls `funcPassByReference(dObj)` and `funcPassByPointer(&dObj)` produce the expected result `Updated _b=-1`.

```
#include <iostream>

class Base {
```

```
public:
    explicit Base(int b) {
        _b = b;
    }
    virtual ~Base() {}
    virtual int update() const;
protected:
    int _b;
};


class Derived: public Base {
public:
    explicit Derived(int b):Base(b) {}
    int update() const;
};

//Class methods definition

int Base::update() const {
    return (_b + 1);
}

int Derived::update() const {
    return (_b -1);
}


//Other function definitions
void funcPassByReference(const Base& bRef) {
    std::cout << "Updated _b=" << bRef.update() << std::endl;
}

void funcPassByPointer(const Base* bPtr) {
    std::cout << "Updated _b=" << bPtr->update() << std::endl;
}

int main() {
    Derived dObj(0);
    funcPassByReference(dObj);       //Function call does not slice object
    funcPassByPointer(&dObj);       //Function call does not slice object
    return 0;
 }
```

Note If you pass by value, because a copy of the object is made, the original object is not modified. Passing by reference or by pointer makes the object vulnerable to modification. If you are concerned about your original object being modified, add a `const` qualifier to your function parameter, as in the preceding example.

Check Information

Group: 09. Object Oriented Programming (OOP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics
“Check for Coding Standard Violations”

External Websites
OOP51-CPP

Introduced in R2019a

CERT C++: OOP52-CPP

Do not delete a polymorphic object without a virtual destructor

Description

Rule Definition

Do not delete a polymorphic object without a virtual destructor.

Polyspace Implementation

This checker checks for **Base class destructor not virtual**.

Examples

Base class destructor not virtual

Issue

Base class destructor not virtual occurs when a class has `virtual` functions but not a `virtual` destructor.

Risk

The presence of `virtual` functions indicates that the class is intended for use as a base class. However, if the class does not have a `virtual` destructor, it cannot behave polymorphically for deletion of derived class objects.

If a pointer to this class refers to a derived class object, and you use the pointer to delete the object, only the base class destructor is called. Additional resources allocated in the derived class are not released and can cause a resource leak.

Fix

One possible fix is to always use a `virtual` destructor in a class that contains `virtual` functions.

Example - Base Class Destructor Not Virtual

```
class Base {
        public:
                Base(): _b(0) {};
                virtual void update() {_b += 1;};
        private:
                int _b;
};

class Derived: public Base {
        public:
                Derived(): _d(0) {};
                ~Derived() {_d = 0;};
                virtual void update() {_d += 1;};
        private:
                int _d;
};
```

In this example, the class `Base` does not have a `virtual` destructor. Therefore, if a `Base*` pointer points to a `Derived` object that is allocated memory dynamically, and the `delete` operation is performed on that `Base*` pointer, the `Base` destructor is called. The memory allocated for the additional member `_d` is not released.

The defect appears on the base class definition. Following are some tips for navigating in the source code:

- To find classes derived from the base class, right-click the base class name and select **Search For All References**. Browse through each search result to find derived class definitions.
- To find if you are using a pointer or reference to a base class to point to a derived class object, right-click the base class name and select **Search For All References**. Browse through search results that start with `Base*` or `Base&` to locate pointers or references to the base class. You can then see if you are using a pointer or reference to point to a derived class object.

Correction — Make Base Class Destructor Virtual

One possible correction is to declare a `virtual` destructor for the class `Base`.

```
class Base {
        public:
                Base(): _b(0) {};
                virtual ~Base() {_b = 0;};
                virtual void update() {_b += 1;};
        private:
                int _b;
};

class Derived: public Base {
        public:
                Derived(): _d(0) {};
                ~Derived() {_d = 0;};
                virtual void update() {_d += 1;};
        private:
                int _d;
};
```

Check Information

Group: 09. Object Oriented Programming (OOP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

OOP52-CPP

Introduced in R2019a

CERT C++: OOP53-CPP

Write constructor member initializers in the canonical order

Description

Rule Definition

Write constructor member initializers in the canonical order.

Polyspace Implementation

This checker checks for **Members not initialized in canonical order**.

Examples

Members not initialized in canonical order

Description

Members not initialized in canonical order occurs when the initializer list of a class constructor:

- Does not initialize data members of the class in the order in which they are declared.

  For instance:

  ```
  class aClass {
     int var1;
     int var2;
  public:
     aClass(int val): var2(val), var1(val) {}
  };
  ```

- Does not call base class constructors in the order in which they appear in the base-specifier list.

  For instance:

  ```
  class aClass: baseClass1, baseClass2 {
     aClass(int val): baseClass2(val), baseClass1(val) {}
  }
  ```

Risk

The order in which data members or base class constructors appear in the initializer list does not reflect the actual order of initialization. Data members are initialized in the order of declaration and base class constructors are called in the order in which they appear in the base-specifier list.

However, you or another developer can mistake the order in the initializer list as the actual initialization order. As a result, you might introduce dependencies between the initializations that results in reading an uninitialized region of memory. For instance, this initializer list might indicate that `bVar` is first initialized with the constructor argument `x` and then `aVar` is initialized with `bVar`:

```
class aClass {
   int aVar;
   int bVar;
```

```
public:
   aClass(int x): bVar(x), aVar(bVar) {}
};
```

However, the initialization happens in the order of declaration and an uninitialized `bVar` is read first.

Fix

In the initializer list of a class constructor:

- Specify class data members in the same order as you declare them in the class

  For instance:

  ```
  class aClass {
     int var1;
     int var2;
  public:
     aClass(int val): var1(val), var2(val) {}
  };
  ```

- Call base constructors in the same order as you specify them in the base-specifier list.

  For instance:

  For instance:

  ```
  class aClass: baseClass1, baseClass2 {
     aClass(int val): baseClass1(val), baseClass2(val) {}
  }
  ```

Check Information

Group: 09. Object Oriented Programming (OOP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

OOP53-CPP

Introduced in R2020a

CERT C++: OOP54-CPP

Gracefully handle self-copy assignment

Description

Rule Definition

Gracefully handle self-copy assignment.

Polyspace Implementation

This checker checks for **Self assignment not tested in operator**.

Examples

Self assignment not tested in operator

Issue

Self assignment not tested in operator occurs when you do not test if the argument to the copy assignment operator of an object is the object itself.

Risk

Self-assignment causes unnecessary copying. Though it is unlikely that you assign an object to itself, because of aliasing, you or users of your class cannot always detect a self-assignment.

Self-assignment can cause subtle errors if a data member is a pointer and you allocate memory dynamically to the pointer. In your copy assignment operator, you typically perform these steps:

1. Deallocate the memory originally associated with the pointer.

   ```
   delete ptr;
   ```

2. Allocate new memory to the pointer. Initialize the new memory location with contents obtained from the operator argument.

   ```
    ptr = new ptrType(*(opArgument.ptr));
   ```

If the argument to the operator, `opArgument`, is the object itself, after your first step, the pointer data member in the operator argument, `opArgument.ptr`, is not associated with a memory location. `*opArgument.ptr` contains unpredictable values. Therefore, in the second step, you initialize the new memory location with unpredictable values.

Fix

Test for self-assignment in the copy assignment operator of your class. Only after the test, perform the assignments in the copy assignment operator.

Example - Missing Test for Self-Assignment

```
class MyClass1 { };
class MyClass2 {
public:
```

```
    MyClass2()                : p_(new MyClass1())      { }
    MyClass2(const MyClass2& f)   : p_(new MyClass1(*f.p_)) { }
    ~MyClass2()               {
        delete p_;
    }
    MyClass2& operator= (const MyClass2& f)
    {
        delete p_;
        p_ = new MyClass1(*f.p_);
        return *this;
    }
private:
    MyClass1* p_;
};
```

In this example, the copy assignment operator in `MyClass2` does not test for self-assignment. If the parameter `f` is the current object, after the statement `delete p_`, the memory allocated to pointer `f.p_` is also deallocated. Therefore, the statement `p_ = new MyClass1(*f.p_)` initializes the memory location that `p_` points to with unpredictable values.

Correction — Test for Self-Assignment

One possible correction is to test for self-assignment in the copy assignment operator.

```
class MyClass1 { };
class MyClass2 {
public:
    MyClass2()                : p_(new MyClass1())      { }
    MyClass2(const MyClass2& f)   : p_(new MyClass1(*f.p_)) { }
    ~MyClass2()               {
        delete p_;
    }
    MyClass2& operator= (const MyClass2& f)
    {
        if(&f != this) {
           delete p_;
           p_ = new MyClass1(*f.p_);
        }
        return *this;
    }
private:
    MyClass1* p_;
};
```

Check Information

Group: 09. Object Oriented Programming (OOP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

OOP54-CPP

Introduced in R2019a

CERT C++: OOP57-CPP

Prefer special member functions and overloaded operators to C Standard Library functions

Description

Rule Definition

Prefer special member functions and overloaded operators to C Standard Library functions.

Polyspace Implementation

This checker checks for **Bytewise operations on nontrivial class object**.

Examples

Bytewise operations on nontrivial class object

Issue

Bytewise operations on nontrivial class object occurs when you use C Standard library functions to perform bytewise operation on non-trivial or non-standard layout class type objects. For definitions of trivial and standard layout classes, see the C++ Standard, [class], paragraphs 6 and 7 respectively.

The checker raises a defect you initialize or copy non-trivial class type objects using these functions:

- `std::memset`
- `std::memcpy`
- `std::strcpy`
- `std::memmove`

Or when you compare non-standard layout class type objects using these functions:

- `std::memcmp`
- `std::strcmp`

Bytewise operations on nontrivial class object raises no defect if the bytewise operation is performed through an alias. For example no defect is raised in the bytewise comparison and copy operations in this code. The bytewise operations use `dptr` and `sptr`, the aliases of non-trivial or non-standard layout class objects `d` and `s`.

```
void func(NonTrivialNonStdLayout *d, const NonTrivialNonStdLayout *s)
{
   void* dptr = (void*)d;
   const void* sptr = (void*)s;
   // ...
   // ...
   // ...
   if (!std::memcmp(dptr, sptr, sizeof(NonTrivialNonStdLayout))) {
     (void)std::memcpy(dptr, sptr, sizeof(NonTrivialNonStdLayout));
      // ...
   }
}
```

Risk

Performing bytewise comparison operations by using C Standard library functions on non-trivial or non-standard layout class type object might result in unexpected values due to implementation details. The object representation depends on the implementation details, such as the order of private and public members, or the use of virtual function pointer tables to represent the object.

Performing bytewise setting operations by using C Standard library functions on non-trivial or non-standard layout class type object can change the implementation details. The operation might result in abnormal program behavior or a code execution vulnerability. For instance, if the address of a member function is overwritten, the call to this function invokes an unexpected function.

Fix

To perform bytewise operations non-trivial or non-standard layout class type object, use these C++ special member functions instead of C Standard library functions.

C Standard Library Functions	C++ Member Functions
`std::memset`	Class constructor

`std::memcpy`
`std::strcpy`
`std::memmove`
	Class copy constructor
	Class move constructor
	Copy assignment operator
	Move assignment operator

`std::memcmp`
`std::strcmp`
	`operator<()`
	`operator>()`
	`operator==()`
	`operator!=()`

Example - Using `memset` with non-trivial class object

```
#include <cstring>
#include <iostream>
#include <utility>

class nonTrivialClass
{
    int scalingFactor;
    int otherData;
public:
    nonTrivialClass() : scalingFactor(1) {}
    void set_other_data(int i);
    int f(int i)
    {
        return i / scalingFactor;
    }
    // ...
};

void func()
{
```

```
    nonTrivialClass c;
    // ... Code that mutates c ...
    std::memset(&c, 0, sizeof(nonTrivialClass));
    std::cout << c.f(100) << std::endl;
}
```

In this example, `func()` uses `std::memset` to reinitialize non-trivial class object `c` after it is first initialized with its default constructor. This bytewise operation might not properly initialize the value representation of `c`.

Correction — Define Function Template That Uses `std::swap`

One possible correction is to define a function template `clear()` that uses `std::swap` to perform a swap operation. The call to `clear()`properly reinitializes object `c` by swapping the contents of `c` and default initialized object `empty`.

```
 #include <cstring>
#include <iostream>
#include <utility>

class nonTrivialClass
{
    int scalingFactor;
    int otherData;
public:
    nonTrivialClass() : scalingFactor(1) {}
    void set_other_data(int i);
    int f(int i)
    {
        return i / scalingFactor;
    }
    // ...
};

template <typename T>
T& clear(T& o)
{
    using std::swap;
    T empty;
    swap(o, empty);
    return o;
}

void func()
{
    nonTrivialClass c;
    // ... Code that mutates c ...

    clear(c);
    std::cout << c.f(100) << std::endl;
}
```

Check Information

Group: Rule 09. Object Oriented Programming (OOP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics
“Check for Coding Standard Violations”

External Websites
OOP57-CPP

Introduced in R2019b

CERT C++: OOP58-CPP

Copy operations must not mutate the source object

Description

Rule Definition

Copy operations must not mutate the source object.

Polyspace Implementation

This checker checks for **Copy operation modifying source operand**.

Examples

Copy operation modifying source operand

Issue

Copy operation modifying source operand occurs when a copy constructor or copy assignment operator modifies a mutable data member of its source operand.

For instance, this copy constructor `A` modifies the data member `m` of its source operand `other`:

```
class A {
  mutable int m;

public:
 ...
  A(const A &other) : m(other.m) {
    other.m = 0; //Modification of source
  }
}
```

Risk

A copy operation with a copy constructor (or copy assignment operator):

```
className new_object = old_object; //Calls copy constructor of className
```

copies its source operand `old_object` to its destination operand `new_object`. After the operation, you expect the destination operand to be a copy of the unmodified source operand. If the source operand is modified during copy, this assumption is violated.

Fix

Do not modify the source operand in the copy operation.

If you are modifying the source operand in a copy constructor to implement a move operation, use a move constructor instead. Move constructors are defined in the C++11 standard and later.

Example - Copy Constructor Modifying Source

```
#include <algorithm>
```

```
#include <vector>

class A {
  mutable int m;

public:
  A() : m(0) {}
  explicit A(int m) : m(m) {}

  A(const A &other) : m(other.m) {
    other.m = 0;
  }

  A& operator=(const A &other) {
    if (&other != this) {
      m = other.m;
      other.m = 0;
    }
    return *this;
  }

  int get_m() const { return m; }
};

void f() {
  std::vector<A> v{10};
  A obj(12);
  std::fill(v.begin(), v.end(), obj);
}
```

In this example, a vector of ten objects of type `A` is created. The `std::fill` function copies an object of type `A`, which has a data member with value 12, to each of the ten objects. After this operation, you might expect that all ten objects in the vector have a data member with value 12.

However, the first copy modifies the data member of the source to the value 0. The remaining nine copies copy this value. After the `std::fill` call, the first object in the vector has a data member with value 12 and the remaining objects have data members with value 0.

Correction — Use Move Constructor for Modifying Source

Do not modify data members of the source operand in a copy constructor or copy assignment operator. If you want your class to have a move operation, use a move constructor instead of a copy constructor.

In this corrected example, the copy constructor and copy assignment operator of class `A` do not modify the data member `m`. A separate move constructor modifies the source operand.

```
#include <algorithm>
#include <vector>

class A {
  int m;

public:
  A() : m(0) {}
  explicit A(int m) : m(m) {}
```

```
  A(const A &other) : m(other.m) {}
  A(A &&other) : m(other.m) { other.m = 0; }

  A& operator=(const A &other) {
    if (&other != this) {
      m = other.m;
    }
    return *this;
  }

  //Move constructor
  A& operator=(A &&other) {
    m = other.m;
    other.m = 0;
    return *this;
  }

  int get_m() const { return m; }
};

void f() {
  std::vector<A> v{10};
  A obj(12);
  std::fill(v.begin(), v.end(), obj);
}
```

Check Information

Group: 09. Object Oriented Programming (OOP)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

OOP58-CPP

Introduced in R2019a

CERT C++: CON33-C

Avoid race conditions when using library functions

Description

Rule Definition

Avoid race conditions when using library functions.

Polyspace Implementation

This checker checks for **Data race through standard library function call**.

Examples

Data race through standard library function call

Issue

Data race through standard library function call occurs when:

1. Multiple tasks call the same standard library function.

   For instance, multiple tasks call the `strerror` function.
2. The calls are not protected using a common protection.

   For instance, the calls are not protected by the same critical section.

Functions flagged by this defect are not guaranteed to be reentrant. A function is reentrant if it can be interrupted and safely called again before its previous invocation completes execution. If a function is not reentrant, multiple tasks calling the function without protection can cause concurrency issues. For the list of functions that are flagged, see CON33-C: Avoid race conditions when using library functions.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

Risk

The functions flagged by this defect are nonreentrant because their implementations can use global or static variables. When multiple tasks call the function without protection, the function call from one task can interfere with the call from another task. The two invocations of the function can concurrently access the global or static variables and cause unpredictable results.

The calls can also cause more serious security vulnerabilities, such as abnormal termination, denial-of-service attack, and data integrity violations.

Fix

To fix this defect, do one of the following:

- Use a reentrant version of the standard library function if it exists.

  For instance, instead of `strerror()`, use `strerror_r()` or `strerror_s()`. For alternatives to functions flagged by this defect, see the documentation for CON33-C.

- Protect the function calls using common critical sections or temporal exclusion.

  See `Critical section details (-critical-section-begin -critical-section-end)` and `Temporally exclusive tasks (-temporal-exclusions-file)`.

  To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the  icon. For an example, see below.

Example - Unprotected Call to Standard Library Function from Multiple Tasks

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno);
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

Option	Specification	
Configure multitasking manually	☑	
Tasks (-entry-points)	task1 task2 task3	
Critical section details (-critical-section-begin -critical-section-end)	**Starting routine**	**Ending routine**
	begin_critical_section	end_critical_section

On the command-line, you can use the following:

```
polyspace-bug-finder
  -entry-points task1,task2,task3
  -critical-section-begin begin_critical_section:cs1
  -critical-section-end end_critical_section:cs1
```

In this example, the tasks, `task1`, `task2` and `task3`, call the function `func`. `func` calls the nonreentrant standard library function, `strerror`.

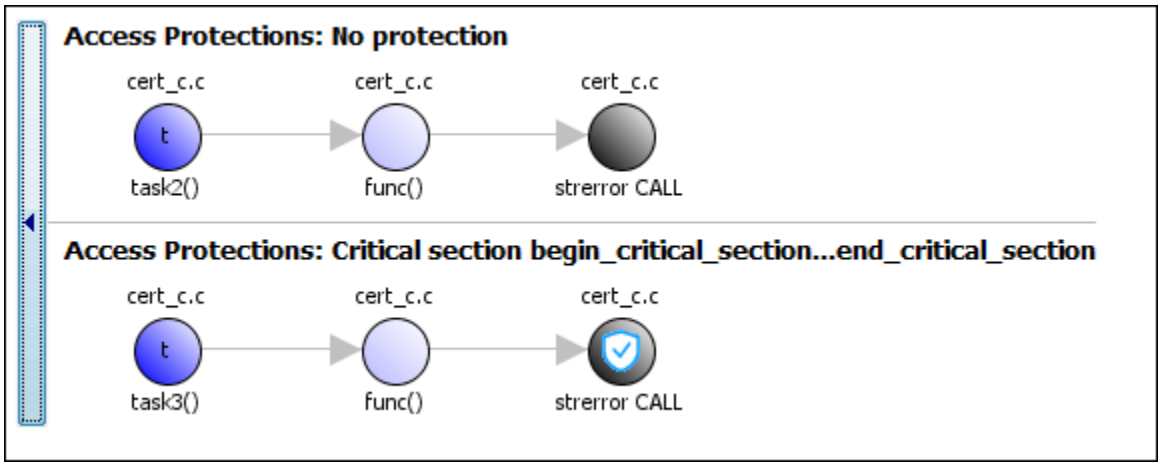
Though `task3` calls `func` inside a critical section, other tasks do not use the same critical section. Operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

These three tasks are calling a nonreentrant standard library function without common protection. In your result details, you see each pair of conflicting function calls.

! Data race through standard library function call (Impact: High)
Certain calls to function 'strerror' can interfere with each other and cause unpredictable results.
To avoid interference, calls to 'strerror' must be in the same critical section.

	Access	Access Protections	Task	File	Scope	Line
	Function call (Non atomic) Operation involves function call	No protection	task1()	data_race_std_lib.c	func()	14
	Function call (Non atomic) Operation involves function call	No protection	task2()	data_race_std_lib.c	func()	14
	Function call (Non atomic) Operation involves function call	No protection	task2()	data_race_std_lib.c	func()	14
	Function call (Non atomic) Operation involves function call	**Critical section begin_critical_section...end_critical_section**	task3()	data_race_std_lib.c	func()	14
	Function call (Non atomic) Operation involves function call	No protection	task1()	data_race_std_lib.c	func()	14
	Function call (Non atomic) Operation involves function call	**Critical section begin_critical_section...end_critical_section**	task3()	data_race_std_lib.c	func()	14

If you click the  icon, you see the function call sequence starting from the entry point to the standard library function call. You also see that the call starting from `task3` is in a critical section. The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.



Correction — Use Reentrant Version of Standard Library Function

One possible correction is to use a reentrant version of the standard library function `strerror`. You can use the POSIX version `strerror_r` which has the same functionality but also guarantees thread-safety.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);
enum { BUFFERSIZE = 64 };

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char errmsg[BUFFERSIZE];
    if (strerror_r(errno, errmsg, BUFFERSIZE) != 0) {
      /* Handle error */
    }
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
```

```
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

Correction — Place Function Call in Critical Section

One possible correction is to place the call to `strerror` in critical section. You can implement the critical section in multiple ways.

For instance, you can place the call to the intermediate function `func` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `func` and therefore the calls to `strerror` from the three tasks cannot interfere with each other.

To implement the critical section, in each of the three tasks, call `func` between calls to `begin_critical_section` and `end_critical_section`.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno);
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    begin_critical_section();
    func(fptr1);
    end_critical_section();
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     begin_critical_section();
     func(fptr2);
     end_critical_section();
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

Option	Value
`Temporally exclusive tasks (-temporal-exclusions-file)`	`task1 task2 task3`

On the command-line, you can use the following:

```
polyspace-bug-finder
    -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

Check Information

Group: 10. Concurrency (CON)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

CON33-C

Introduced in R2019a

CERT C++: CON37-C

Do not call signal() in a multithreaded program

Description

Rule Definition

Do not call signal() in a multithreaded program.

Polyspace Implementation

This checker checks for **Signal call in multithreaded program**.

Examples

Signal call in multithreaded program

Issue

Signal call in multithreaded program occurs when you use the `signal()` function in a program with multiple threads.

Risk

According to the C11 standard (Section 7.14.1.1), use of the `signal()` function in a multithreaded program is undefined behavior.

Fix

Depending on your intent, use other ways to perform an asynchronous action on a specific thread.

Example - Use of `signal()` Function to Terminate Loop in Thread

```
#include <signal.h>
#include <stddef.h>
#include <threads.h>

volatile sig_atomic_t flag = 0;

void handler(int signum) {
  flag = 1;
}

/* Runs until user sends SIGUSR1 */
int func(void *data) {
  while (!flag) {
    /* ... */
  }
  return 0;
}

int main(void) {
  signal(SIGINT, handler); /* Undefined behavior */
  thrd_t tid;
```

```
  if (thrd_success != thrd_create(&tid, func, NULL)) {
    /* Handle error */
  }
  /* ... */
  return 0;
}
```

In this example, the `signal` function is used to terminate a `while` loop in the thread created with `thrd_create`.

Correction — Use `atomic_bool` Variable to Terminate Loop

One possible correction is to use an `atomic_bool` variable that multiple threads can access. In the corrected example, the child thread evaluates this variable before every loop iteration. After completing the program, you can modify this variable so that the child thread exits the loop.

```
#include <stdatomic.h>
#include <stdbool.h>
#include <stddef.h>
#include <threads.h>

atomic_bool flag = ATOMIC_VAR_INIT(false);

int func(void *data) {
  while (!flag) {
    /* ... */
  }
  return 0;
}

int main(void) {
  thrd_t tid;

  if (thrd_success != thrd_create(&tid, func, NULL)) {
    /* Handle error */
  }
  /* ... */
  /* Set flag when done */
  flag = true;

  return 0;
}
```

Check Information

Group: 10. Concurrency (CON)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

CON37-C

Introduced in R2019a

CERT C++: CON40-C

Do not refer to an atomic variable twice in an expression

Description

Rule Definition

Do not refer to an atomic variable twice in an expression.

Polyspace Implementation

This checker checks for these issues:

- **Atomic variable accessed twice in an expression**.
- **Atomic load and store sequence not atomic**.

Examples

Atomic variable accessed twice in an expression

Issue

Atomic variable accessed twice in an expression occurs when C atomic types or C++ `std::atomic` class variables appear twice in an expression and there are:

- Two atomic read operations on the variable.
- An atomic read and a distinct atomic write operation on the variable.

The C standard defines certain operations on atomic variables that are thread safe and do not cause data race conditions. Unlike individual operations, a pair of operations on the same atomic variable in an expression is not thread safe.

Risk

A thread can modify the atomic variable between the pair of atomic operations, which can result in a data race condition.

Fix

Do not reference an atomic variable twice in the same expression.

Example - Referencing Atomic Variable Twice in an Expression

```
#include <stdatomic.h>

atomic_int n = ATOMIC_VAR_INIT(0);

int compute_sum(void)
{
    return n * (n + 1) / 2;
}
```

In this example, the global variable `n` is referenced twice in the return statement of `compute_sum()`. The value of `n` can change between the two distinct read operations. `compute_sum()` can return an incorrect value.

Correction — Pass Variable as Function Argument

One possible correction is to pass the variable as a function argument `n`. The variable is copied to memory and the read operations on the copy guarantee that `compute_sum()` returns a correct result. If you pass a variable of type `int` instead of type `atomic_int`, the correction is still valid.

```
#include <stdatomic.h>

int compute_sum(atomic_int n)
{
    return n * (n + 1) / 2;
}
```

Atomic load and store sequence not atomic

Issue

Atomic load and store sequence not atomic occurs when you use these functions to load, and then store an atomic variable.

- C functions:
  - `atomic_load()`
  - `atomic_load_explicit()`
  - `atomic_store()`
  - `atomic_store_explicit()`
- C++ functions:
  - `std::atomic_load()`
  - `std::atomic_load_explicit()`
  - `std::atomic_store()`
  - `std::atomic_store_explicit()`
  - `std::atomic::load()`
  - `std::atomic::store()`

A thread cannot interrupt an atomic load or an atomic store operation on a variable, but a thread can interrupt a store, and then load sequence.

Risk

A thread can modify a variable between the load and store operations, resulting in a data race condition.

Fix

To read, modify, and store a variable atomically, use a compound assignment operator such as `+=`, `atomic_compare_exchange()` or `atomic_fetch_*`-family functions.

Example - Loading Then Storing an Atomic Variable

```
#include <stdatomic.h>
#include <stdbool.h>
```

```
static atomic_bool flag = ATOMIC_VAR_INIT(false);

void init_flag(void)
{
    atomic_init(&flag, false);
}

void toggle_flag(void)
{
    bool temp_flag = atomic_load(&flag);
    temp_flag = !temp_flag;
    atomic_store(&flag, temp_flag);
}

bool get_flag(void)
{
    return atomic_load(&flag);
}
```

In this example, variable `flag` of type `atomic_bool` is referenced twice inside the `toggle_flag()` function. The function loads the variable, negates its value, then stores the new value back to the variable. If two threads call `toggle_flag()`, the second thread can access `flag` between the load and store operations of the first thread. `flag` can end up in an incorrect state.

Correction — Use Compound Assignment to Modify Variable

One possible correction is to use a compound assignment operator to toggle the value of `flag`. The C standard defines the operation by using ^= as atomic.

```
 #include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void toggle_flag(void)
{
    flag ^= 1;
}

bool get_flag(void)
{
    return flag;
}
```

Check Information

Group: 10. Concurrency (CON)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

CON40-C

Introduced in R2019a

CERT C++: CON41-C

Wrap functions that can fail spuriously in a loop

Description

Rule Definition

Wrap functions that can fail spuriously in a loop.

Polyspace Implementation

This checker checks for **Function that can spuriously fail not wrapped in loop**.

Examples

Function that can spuriously fail not wrapped in loop

Issue

Function that can spuriously fail not wrapped in loop occurs when the following atomic compare and exchange functions that can fail spuriously are called from outside a loop.

- C atomic functions:
  - `atomic_compare_exchange_weak()`
  - `atomic_compare_exchange_weak_explicit()`
- C++ atomic functions:
  - `std::atomic<T>::compare_exchange_weak(T* expected, T desired)`
  - `std::atomic<T>::compare_exchange_weak_explicit(T* expected, T desired, std::memory_order succ, std::memory_order fail)`
  - `std::atomic_compare_exchange_weak(std::atomic<T>* obj, T* expected, T desired)`
  - `std::atomic_compare_exchange_weak_explicit(volatile std::atomic<T>* obj, T* expected, T desired, std::memory_order succ, std::memory_order fail)`

The functions compare the memory contents of the object representations pointed to by `obj` and `expected`. The comparison can spuriously return false even if the memory contents are equal. This spurious failure makes the functions faster on some platforms.

Risk

An atomic compare and exchange function that spuriously fails can cause unexpected results and unexpected control flow.

Fix

Wrap atomic compare and exchange functions that can spuriously fail in a loop. The loop checks the failure condition after a possible spurious failure.

Example - `atomic_compare_exchange_weak()` Not Wrapped in Loop

```
#include <stdatomic.h>

extern void reset_count(void);
atomic_int count = ATOMIC_VAR_INIT(0);

void increment_count(void)
{
    int old_count = atomic_load(&count);
    int new_count;
    new_count = old_count + 1;
    if (!atomic_compare_exchange_weak(&count, &old_count, new_count))
        reset_count();

}
```

In this example, `increment_count()` uses `atomic_compare_exchange_weak()` to compare `count` and `old_count`. If the counts are equal, `count` is incremented to `new_count`. If they are not equal, the count is reset. When `atomic_compare_exchange_weak()` fails spuriously, the count is reset unnecessarily.

Correction — Wrap `atomic_compare_exchange_weak()` in a `while` Loop

One possible correction is to wrap the call to `atomic_compare_exchange_weak()` in a `while` loop. The loop checks the failure condition after a possible spurious failure.

```
#include <stdatomic.h>

extern void reset_count(void);
atomic_int count = ATOMIC_VAR_INIT(0);

void increment_count(void)
{
    int old_count = atomic_load(&count);
    int new_count;
    new_count = old_count + 1;

    do {
        reset_count();

    } while (!atomic_compare_exchange_weak(&count, &old_count, new_count));

}
```

Check Information

Group: 10. Concurrency (CON)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

CON41-C

Introduced in R2019a

CERT C++: CON43-C

Do not allow data races in multithreaded code

Description

Rule Definition

Do not allow data races in multithreaded code.

Polyspace Implementation

This checker checks for **Data race**.

Examples

Data race

Issue

Data race occurs when:

1. Multiple tasks perform unprotected operations on a shared variable.
2. At least one task performs a write operation.
3. At least one operation is nonatomic. For data race on both atomic and nonatomic operations, see `Data race including atomic operations`.

   See “Define Atomic Operations in Multitasking Code”.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

Risk

Data race can result in unpredictable values of the shared variable because you do not control the order of the operations in different tasks.

Data races between two write operations are more serious than data races between a write and read operation. Two write operations can interfere with each other and result in indeterminate values. To identify write-write conflicts, use the filters on the **Detail** column of the **Results List** pane. For these conflicts, the **Detail** column shows the additional line:

```
Variable value may be altered by write-write concurrent access.
```

See “Filter and Group Results”.

Fix

To fix this defect, protect the operations on the shared variable using critical sections, temporal exclusion or another means. See “Protections for Shared Variables in Multitasking Code”.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing

protections on the calls. To see the function call sequence leading to the conflicts, click the  icon. For an example, see below.

Example - Unprotected Operation on Global Variable from Multiple Tasks

```
int var;
void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
    var++;
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

Option	Specification	
Configure multitasking manually	☑	
Tasks (-entry-points)	task1 task2 task3	
Critical section details (-critical-section-begin -critical-section-end)	**Starting routine**	**Ending routine**
	begin_critical_section	end_critical_section

On the command-line, you can use the following:

```
polyspace-bug-finder
  -entry-points task1,task2,task3
  -critical-section-begin begin_critical_section:cs1
  -critical-section-end end_critical_section:cs1
```

In this example, the tasks `task1`, `task2`, and `task3` call the function `increment`. `increment` contains the operation `var++` that can involve multiple machine instructions including:

- Reading `var`.
- Writing an increased value to `var`.

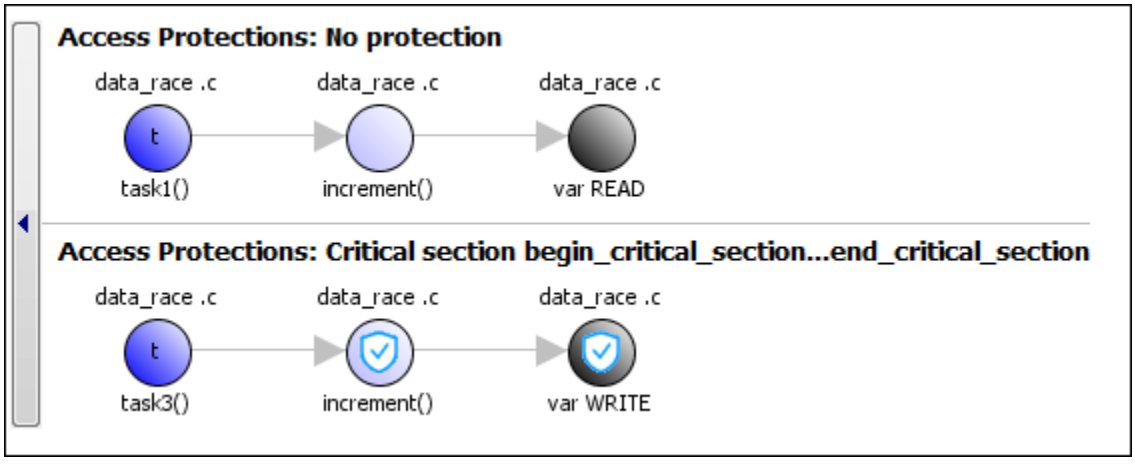
These machine instructions, when executed from `task1` and `task2`, can occur concurrently in an unpredictable sequence. For example, reading `var` from `task1` can occur either before or after writing to `var` from `task2`. Therefore the value of `var` can be unpredictable.

Though `task3` calls `increment` inside a critical section, other tasks do not use the same critical section. The operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

Therefore, the three tasks are operating on a shared variable without common protection. In your result details, you see each pair of conflicting function calls.

	Access	Access Protections	Task	File
	Read	No protection	task1()	data_race .c
	Write (Non atomic) Operation might involve multiple machine instructions	No protection	task2()	data_race .c
	Read	No protection	task1()	data_race .c
	Write (Non atomic) Operation might involve multiple machine instructions	**Critical section begin_critical_section...end_critical_section**	task3()	data_race .c
	Read	No protection	task2()	data_race .c
	Write (Non atomic) Operation might involve multiple machine instructions	**Critical section begin_critical_section...end_critical_section**	task3()	data_race .c

If you click the  icon, you see the function call sequence starting from the entry point to the read or write operation. You also see that the operation starting from `task3` is in a critical section. The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.



Correction — Place Operation in Critical Section

One possible correction is to place the operation in critical section. You can implement the critical section in multiple ways. For instance:

- You can place `var++` in a critical section. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The operation `var++` from the three tasks cannot interfere with each other.

To implement the critical section, in the function `increment`, place the operation `var++` between calls to `begin_critical_section` and `end_critical_section`.

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
      begin_critical_section();
      var++;
      end_critical_section();
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
      increment();
}
```

- You can place the call to `increment` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `increment` from the three tasks cannot interfere with each other.

  To implement the critical section, in each of the three tasks, call `increment` between calls to `begin_critical_section` and `end_critical_section`.

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
      var++;
}

void task1(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}

void task2(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
```

```
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

Option	Value
`Temporally exclusive tasks (-temporal-exclusions-file)`	`task1 task2 task3`

On the command-line, you can use the following:

```
 polyspace-bug-finder
     -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

Example - Unprotected Operation in Threads Created with `pthread_create`

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;


void* increment_count(void* args)
{
    count = count + 1;
    return NULL;
}

void* set_count(void *args)
{
    long long c;
    c = count;
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);
```

```
    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

In this example, Bug Finder detects the creation of separate threads with `pthread_create`. The **Data race** defect is raised because the operation `count = count + 1` in the thread with id `thread_increment` conflicts with the operation `c = count` in the thread with id `thread_get`. The variable `count` is accessed in multiple threads without a common protection.

The two conflicting operations are nonatomic. The operation `c = count` is nonatomic on 32-bit targets. See “Define Atomic Operations in Multitasking Code”.

Correction — Protect Operations with `pthread_mutex_lock` and `pthread_mutex_unlock` Pair

To prevent concurrent access on the variable `count`, protect operations on `count` with a critical section. Use the functions `pthread_mutex_lock` and `pthread_mutex_unlock` to implement the critical section.

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;


void* increment_count(void* args)
{
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}

void* set_count(void *args)
{
    long long c;
    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);

    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

Check Information

Group: 10. Concurrency (CON)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

CON43-C

Introduced in R2019a

CERT C++: CON50-CPP

Do not destroy a mutex while it is locked

Description

Rule Definition

Do not destroy a mutex while it is locked.

Polyspace Implementation

This checker checks for **Destruction of locked mutex**.

Examples

Destruction of locked mutex

Issue

Destruction of locked mutex occurs when a task destroys a mutex after it is locked (and before it is unlocked). The locking and destruction can happen in the same task or different tasks.

Risk

A mutex is locked to protect shared variables from concurrent access. If a mutex is destroyed in the locked state, the protection does not apply.

Fix

To fix this defect, destroy the mutex only after you unlock it. It is a good design practice to:

- Initialize a mutex *before* creating the threads where you use the mutex.
- Destroy a mutex *after* joining the threads that you created.

On the **Result Details** pane, you see two events, the locking and destruction of the mutex, and the tasks that initiated the events. To navigate to the corresponding line in your source code, click the event.

Example - Locking and Destruction in Different Tasks

```
#include <pthread.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;
pthread_mutex_t lock3;

void t0 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
  pthread_mutex_lock (&lock3);
  pthread_mutex_unlock (&lock2);
```

```
  pthread_mutex_unlock (&lock1);
  pthread_mutex_unlock (&lock3);
}

void t1 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
  pthread_mutex_destroy (&lock3);
  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}
```

In this example, after task `t0` locks the mutex `lock3`, task `t1` can destroy it. The destruction occurs if the following events happen in sequence:

1. `t0` acquires `lock3`.
2. `t0` releases `lock2`.
3. `t0` releases `lock1`.
4. `t1` acquires the lock `lock1` released by `t0`.
5. `t1` acquires the lock `lock2` released by `t0`.
6. `t1` destroys `lock3`.

For simplicity, this example uses a mix of automatic and manual concurrency detection. The tasks `t0` and `t1` are manually specified as entry points by using the option `Tasks (-entry-points)`.The critical sections are implemented through primitives `pthread_mutex_lock` and `pthread_mutex_unlock` that the software detects automatically. In practice, for entry point specification (thread creation), you will use primitives such as `pthread_create`. The next example shows how the defect can appear when you use `pthread_create`.

Correction — Place Lock-Unlock Pair Together in Same Critical Section as Destruction

The locking and destruction of `lock3` occurs inside the critical section imposed by `lock1` and `lock2`, but the unlocking occurs outside. One possible correction is to place the lock-unlock pair in the same critical section as the destruction of the mutex. Use one of these critical sections:

- Critical section imposed by `lock1` alone.
- Critical section imposed by `lock1` and `lock2`.

In this corrected code, the lock-unlock pair and the destruction is placed in the critical section imposed by `lock1` and `lock2`. When `t0` acquires `lock1` and `lock2`, `t1` has to wait for their release before it executes the instruction `pthread_mutex_destroy (&lock3);`. Therefore, `t1` cannot destroy mutex `lock3` in the locked state.

```
#include <pthread.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;
pthread_mutex_t lock3;

void t0 (void) {
  pthread_mutex_lock (&lock1);
```

```
  pthread_mutex_lock (&lock2);

  pthread_mutex_lock (&lock3);
  pthread_mutex_unlock (&lock3);

  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}

void t1 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);

  pthread_mutex_destroy (&lock3);

  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}
```

Example - Locking and Destruction in Start Routine of Thread

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 4
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
void atomic_operation(void);

void *do_create(void *arg) {
    /* Creation thread */
    pthread_mutex_init(&lock, NULL);
    pthread_exit((void*) 0);
}

void *do_work(void *arg) {
    /* Worker thread */
    pthread_mutex_lock (&lock);
    atomic_operation();
    pthread_mutex_unlock (&lock);
    pthread_exit((void*) 0);
}

void *do_destroy(void *arg) {
    /* Destruction thread */
    pthread_mutex_destroy(&lock);
    pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
```

```
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Thread that initializes mutex */
   pthread_create(&callThd[0], &attr, do_create, NULL);

   /* Threads that use mutex for atomic operation*/
   for(i=0; i<NUMTHREADS-1; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   /* Thread that destroys mutex */
   pthread_create(&callThd[NUMTHREADS -1], &attr, do_destroy, NULL);

   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   pthread_exit(NULL);
}
```

In this example, four threads are created. The threads are assigned different actions.

- The first thread `callThd[0]` initializes the mutex `lock`.
- The second and third threads, `callThd[1]` and `callThd[2]`, perform an atomic operation protected by the mutex `lock`.
- The fourth thread `callThd[3]` destroys the mutex `lock`.

The threads can interrupt each other. Therefore, immediately after the second or third thread locks the mutex, the fourth thread can destroy it.

Correction — Initialize and Destroy Mutex Outside Start Routine

One possible correction is to initialize and destroy the mutex in the `main` function outside the start routine of the threads. The threads perform only the atomic operation. You need two fewer threads because the mutex initialization and destruction threads are not required.

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 2
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
void atomic_operation(void);

void *do_work(void *arg) {
   pthread_mutex_lock (&lock);
   atomic_operation();
   pthread_mutex_unlock (&lock);
   pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   int i;
   void *status;
```

```
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Initialize mutex */
   pthread_mutex_init(&lock, NULL);

   for(i=0; i<NUMTHREADS; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* Destroy mutex */
   pthread_mutex_destroy(&lock);

   pthread_exit(NULL);
}
```

Correction — Use A Second Mutex To Protect Lock-Unlock Pair and Destruction

Another possible correction is to use a second mutex and protect the lock-unlock pair from the destruction. This corrected code uses the mutex `lock2` to achieve this protection. The second mutex is initialized in the `main` function outside the start routine of the threads.

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 4
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
pthread_mutex_t lock2;
void atomic_operation(void);

void *do_create(void *arg) {
    /* Creation thread */
    pthread_mutex_init(&lock, NULL);
    pthread_exit((void*) 0);
}

void *do_work(void *arg) {
    /* Worker thread */
    pthread_mutex_lock (&lock2);
    pthread_mutex_lock (&lock);
    atomic_operation();
    pthread_mutex_unlock (&lock);
    pthread_mutex_unlock (&lock2);
    pthread_exit((void*) 0);
}
```

```
void *do_destroy(void *arg) {
    /* Destruction thread */
    pthread_mutex_lock (&lock2);
    pthread_mutex_destroy(&lock);
    pthread_mutex_unlock (&lock2);
    pthread_exit((void*) 0);
}


int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Initialize second mutex */
   pthread_mutex_init(&lock2, NULL);

   /* Thread that initializes first mutex */
   pthread_create(&callThd[0], &attr, do_create, NULL);

   /* Threads that use first mutex for atomic operation */
   /* The threads use second mutex to protect first from destruction in locked state*/
   for(i=0; i<NUMTHREADS-1; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   /* Thread that destroys first mutex */
   /* The thread uses the second mutex to prevent destruction of locked mutex */
   pthread_create(&callThd[NUMTHREADS -1], &attr, do_destroy, NULL);


   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* Destroy second mutex */
   pthread_mutex_destroy(&lock2);

   pthread_exit(NULL);
}
```

Check Information

Group: 10. Concurrency (CON)

See Also

Check SEI CERT-C++ (-cert-cpp))

Topics
“Check for Coding Standard Violations”

External Websites
CON50-CPP

Introduced in R2019a

CERT C++: CON52-CPP

Prevent data races when accessing bit-fields from multiple threads

Description

Rule Definition

Prevent data races when accessing bit-fields from multiple threads.

Polyspace Implementation

This checker checks for **Data race**.

Examples

Data race

Issue

Data race occurs when:

1. Multiple tasks perform unprotected operations on a shared variable.
2. At least one task performs a write operation.
3. At least one operation is nonatomic. For data race on both atomic and nonatomic operations, see `Data race including atomic operations`.

   See “Define Atomic Operations in Multitasking Code”.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

Risk

Data race can result in unpredictable values of the shared variable because you do not control the order of the operations in different tasks.

Data races between two write operations are more serious than data races between a write and read operation. Two write operations can interfere with each other and result in indeterminate values. To identify write-write conflicts, use the filters on the **Detail** column of the **Results List** pane. For these conflicts, the **Detail** column shows the additional line:

```
Variable value may be altered by write-write concurrent access.
```

See “Filter and Group Results”.

Fix

To fix this defect, protect the operations on the shared variable using critical sections, temporal exclusion or another means. See “Protections for Shared Variables in Multitasking Code”.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing

protections on the calls. To see the function call sequence leading to the conflicts, click the  icon. For an example, see below.

Example - Unprotected Operation on Global Variable from Multiple Tasks

```
int var;
void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
    var++;
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

Option	Specification	
Configure multitasking manually	☑	
Tasks (-entry-points)	task1 task2 task3	
Critical section details (-critical-section-begin -critical-section-end)	**Starting routine**	**Ending routine**
	begin_critical_section	end_critical_section

On the command-line, you can use the following:

```
 polyspace-bug-finder
   -entry-points task1,task2,task3
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

In this example, the tasks `task1`, `task2`, and `task3` call the function `increment`. `increment` contains the operation `var++` that can involve multiple machine instructions including:

- Reading `var`.
- Writing an increased value to `var`.

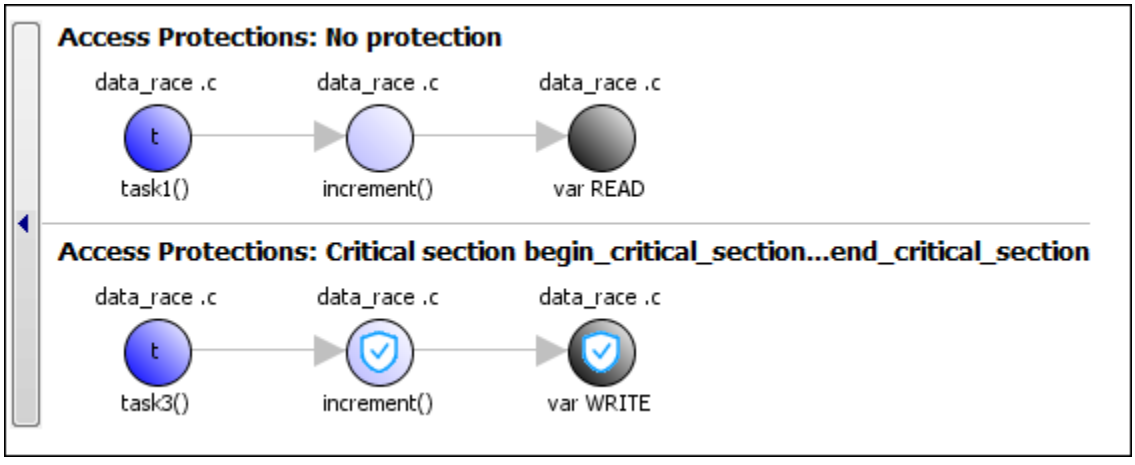
These machine instructions, when executed from `task1` and `task2`, can occur concurrently in an unpredictable sequence. For example, reading `var` from `task1` can occur either before or after writing to `var` from `task2`. Therefore the value of `var` can be unpredictable.

Though `task3` calls `increment` inside a critical section, other tasks do not use the same critical section. The operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

Therefore, the three tasks are operating on a shared variable without common protection. In your result details, you see each pair of conflicting function calls.

	Access	Access Protections	Task	File
	Read	No protection	task1()	data_race .c
	Write (Non atomic) Operation might involve multiple machine instructions	No protection	task2()	data_race .c
	Read	No protection	task1()	data_race .c
	Write (Non atomic) Operation might involve multiple machine instructions	**Critical section begin_critical_section...end_critical_section**	task3()	data_race .c
	Read	No protection	task2()	data_race .c
	Write (Non atomic) Operation might involve multiple machine instructions	**Critical section begin_critical_section...end_critical_section**	task3()	data_race .c

If you click the  icon, you see the function call sequence starting from the entry point to the read or write operation. You also see that the operation starting from `task3` is in a critical section. The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.



Correction — Place Operation in Critical Section

One possible correction is to place the operation in critical section. You can implement the critical section in multiple ways. For instance:

- You can place `var++` in a critical section. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The operation `var++` from the three tasks cannot interfere with each other.

To implement the critical section, in the function `increment`, place the operation `var++` between calls to `begin_critical_section` and `end_critical_section`.

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
      begin_critical_section();
      var++;
      end_critical_section();
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
      increment();
}
```

- You can place the call to `increment` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `increment` from the three tasks cannot interfere with each other.

  To implement the critical section, in each of the three tasks, call `increment` between calls to `begin_critical_section` and `end_critical_section`.

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
      var++;
}

void task1(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}

void task2(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
```

```
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

Option	Value
`Temporally exclusive tasks (-temporal-exclusions-file)`	`task1 task2 task3`

On the command-line, you can use the following:

```
 polyspace-bug-finder
     -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

Example - Unprotected Operation in Threads Created with `pthread_create`

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;


void* increment_count(void* args)
{
    count = count + 1;
    return NULL;
}

void* set_count(void *args)
{
    long long c;
    c = count;
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);
```

```
    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

In this example, Bug Finder detects the creation of separate threads with `pthread_create`. The **Data race** defect is raised because the operation `count = count + 1` in the thread with id `thread_increment` conflicts with the operation `c = count` in the thread with id `thread_get`. The variable `count` is accessed in multiple threads without a common protection.

The two conflicting operations are nonatomic. The operation `c = count` is nonatomic on 32-bit targets. See “Define Atomic Operations in Multitasking Code”.

Correction — Protect Operations with `pthread_mutex_lock` and `pthread_mutex_unlock` Pair

To prevent concurrent access on the variable `count`, protect operations on `count` with a critical section. Use the functions `pthread_mutex_lock` and `pthread_mutex_unlock` to implement the critical section.

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;


void* increment_count(void* args)
{
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}

void* set_count(void *args)
{
    long long c;
    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);

    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

Check Information

Group: 10. Concurrency (CON)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

CON52-CPP

Introduced in R2019a

CERT C++: CON53-CPP

Avoid deadlock by locking in a predefined order

Description

Rule Definition

Avoid deadlock by locking in a predefined order.

Polyspace Implementation

This checker checks for **Deadlock**.

Examples

Deadlock

Issue

Deadlock occurs when multiple tasks are stuck in their critical sections (CS) because:

- Each CS waits for another CS to end.
- The critical sections (CS) form a closed cycle. For example:
  - CS #1 waits for CS #2 to end, and CS #2 waits for CS #1 to end.
  - CS #1 waits for CS #2 to end, CS #2 waits for CS #3 to end and CS #3 waits for CS #1 to end.

Polyspace expects critical sections of code to follow a specific format. A critical section lies between a call to a lock function and a call to an unlock function. When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait until `my_task` calls the corresponding unlock function. Both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Risk

Each task waits for a critical section in another task to end and is unable to proceed. The program can freeze indefinitely.

Fix

The fix depends on the root cause of the defect. You can try to break the cyclic order between the tasks in one of these ways:

- Write down all critical sections involved in the deadlock in a certain sequence. Whenever you call the lock functions of the critical sections within a task, respect the order in that sequence. See an example below.
- If one of the critical sections involved in a deadlock occurs in an interrupt, try to disable all interrupts during critical sections in all tasks. See `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Reviewing this defect is an opportunity to check if all operations in your critical section are really meant to be executed as an atomic block. It is a good practice to keep critical sections at a bare minimum.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Deadlock with Two Tasks

```
void task1(void);
void task2(void);

int var;
void perform_task_cycle(void) {
 var++;
}

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_2();
    begin_critical_section_1();
    perform_task_cycle();
    end_critical_section_1();
    end_critical_section_2();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Specification
Configure multitasking manually	☑

Entry points	`task1`
	
	`task2`

Option	Specification	
Critical section details	**Starting routine**	**Ending routine**
	`begin_critical_section_1`	`end_critical_section_1`
	`begin_critical_section_2`	`end_critical_section_2`

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. `task1` calls `begin_critical_section_1`.
2. `task2` calls `begin_critical_section_2`.
3. `task1` reaches the instruction `begin_critical_section_2();`. Since `task2` has already called `begin_critical_section_2`, `task1` waits for `task2` to call `end_critical_section_2`.
4. `task2` reaches the instruction `begin_critical_section_1();`. Since `task1` has already called `begin_critical_section_1`, `task2` waits for `task1` to call `end_critical_section_1`.

Correction-Follow Same Locking Sequence in Both Tasks

One possible correction is to follow the same sequence of calls to lock and unlock functions in both `task1` and `task2`.

```
void task1(void);
void task2(void);
void perform_task_cycle(void);

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}
```

Example - Deadlock with More Than Two Tasks

```
int var;
void performTaskCycle() {
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);


void unlock1(void);
void unlock2(void);
void unlock3(void);

void task1() {
 while(1) {
    lock1();
    lock2();
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock3();
    lock1();
    performTaskCycle();
    unlock1();
    unlock3();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Specification
Configure multitasking manually	☑

Option	Specification	

Entry points	`task1`
	`task2`
	`task3`	

Critical section details	**Starting routine**	**Ending routine**
	`lock1`	`unlock1`
	`lock2`	`unlock2`
	`lock3`	`unlock3`

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. `task1` calls `lock1`.
2. `task2` calls `lock2`.
3. `task3` calls `lock3`.
4. `task1` reaches the instruction `lock2();`. Since `task2` has already called `lock2`, `task1` waits for call to `unlock2`.
5. `task2` reaches the instruction `lock3();`. Since `task3` has already called `lock3`, `task2` waits for call to `unlock3`.
6. `task3` reaches the instruction `lock1();`. Since `task1` has already called `lock1`, `task3` waits for call to `unlock1`.

Correction — Break Cyclic Order

To break the cyclic order between critical sections, note every lock function in your code in a certain sequence, for example:

1. `lock1`
2. `lock2`
3. `lock3`

If you use more than one lock function in a task, use them in the order in which they appear in the sequence. For example, you can use `lock1` followed by `lock2` but not `lock2` followed by `lock1`.

```
int var;
void performTaskCycle() {
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);

void unlock1(void);
void unlock2(void);
void unlock3(void);
```

```
void task1() {
 while(1) {
    lock1();
    lock2();
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock1();
    lock3();
    performTaskCycle();
    unlock3();
    unlock1();
 }
}
```

Check Information

Group: 10. Concurrency (CON)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

CON53-CPP

Introduced in R2019a

CERT C++: CON54-CPP

Wrap functions that can spuriously wake up in a loop

Description

Rule Definition

Wrap functions that can spuriously wake up in a loop.

Polyspace Implementation

This checker checks for **Function that can spuriously wake up not wrapped in loop**.

Examples

Function that can spuriously wake up not wrapped in loop

Issue

Function that can spuriously wake up not wrapped in loop occurs when the following wait-on-condition functions are called from outside a loop:

- C functions:
  - `cnd_wait()`
  - `cnd_timedwait()`
- POSIX functions:
  - `pthread_cond_wait()`
  - `pthread_cond_timedwait()`
- C++ `std::condition_variable` and `std::condition_variable_any` class member functions:
  - `wait()`
  - `wait_until()`
  - `wait_for()`

Wait-on-condition functions pause the execution of the calling thread when a specified condition is met. The thread wakes up and resumes once another thread notifies it with `cnd_broadcast()` or an equivalent function. The wake-up notification can be spurious or malicious.

Risk

If a thread receives a spurious wake-up notification and the condition of the wait-on-condition function is not checked, the thread can wake up prematurely. The wake-up can cause unexpected control flow, indefinite blocking of other threads, or denial of service.

Fix

Wrap wait-on-condition functions that can wake up spuriously in a loop. The loop checks the wake-up condition after a possible spurious wake-up notification.

Example - `cnd_wait()` Not Wrapped in Loop

```
#include <stdio.h>
#include <stddef.h>
#include <threads.h>

#define THRESHOLD 100

static mtx_t lock;
static cnd_t cond;

void func(int input)
{
    if (thrd_success != mtx_lock(&lock)) {
        /* Handle error */
    }
    /* test condition to pause thread */
    if (input > THRESHOLD) {
        if (thrd_success != cnd_wait(&cond, &lock)) {
            /* Handle error */
        }
    }
    /* Proceed if condition to pause does not hold */


    if (thrd_success != mtx_unlock(&lock)) {
        /* Handle error */
    }
}
```

In this example, the thread uses `cnd_wait()` to pause execution when `input` is greater than `THRESHOLD`. The paused thread can resume if another thread uses `cnd_broadcast()`, which notifies all the threads. This notification causes the thread to wake up even if the pause condition is still true.

Correction — Wrap `cnd_wait()` in a `while` Loop

One possible correction is to wrap `cnd_wait()` in a `while` loop. The loop checks the pause condition after the thread receives a possible spurious wake-up notification.

```
#include <stdio.h>
#include <stddef.h>
#include <threads.h>

#define THRESHOLD 100

static mtx_t lock;
static cnd_t cond;

void func(int input)
{
    if (thrd_success != mtx_lock(&lock)) {
        /* Handle error */
    }
    /* test condition to pause thread */
    while (input > THRESHOLD) {
        if (thrd_success != cnd_wait(&cond, &lock)) {
            /* Handle error */
```

```
        }
    }
    /* Proceed if condition to pause does not hold */


    if (thrd_success != mtx_unlock(&lock)) {
        /* Handle error */
    }
}
```

Check Information

Group: 10. Concurrency (CON)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

CON54-CPP

Introduced in R2019a

CERT C++: ENV30-C

Do not modify the object referenced by the return value of certain functions

Description

Rule Definition

Do not modify the object referenced by the return value of certain functions.

Polyspace Implementation

This checker checks for **Modification of internal buffer returned from nonreentrant standard function**.

Examples

Modification of internal buffer returned from nonreentrant standard function

Issue

Modification of internal buffer returned from nonreentrant standard function occurs when the following happens:

- A nonreentrant standard function returns a pointer.
- You attempt to write to the memory location that the pointer points to.

Nonreentrant standard functions that return a non `const`-qualified pointer to an internal buffer include `getenv`, `getlogin`, `crypt`, `setlocale`, `localeconv`, `strerror` and others.

Risk

Modifying the internal buffer that a nonreentrant standard function returns can cause the following issues:

- It is possible that the modification does not succeed or alters other internal data.

  For instance, `getenv` returns a pointer to an environment variable value. If you modify this value, you alter the environment of the process and corrupt other internal data.
- Even if the modification succeeds, it is possible that a subsequent call to the same standard function does not return your modified value.

  For instance, you modify the environment variable value that `getenv` returns. If another process, thread, or signal handler calls `setenv`, the modified value is overwritten. Therefore, a subsequent call to `getenv` does not return your modified value.

Fix

Avoid modifying the internal buffer using the pointer returned from the function.

Example - Modification of `getenv` Return Value

```
#include <stdlib.h>
#include <string.h>
```

```
void printstr(const char*);

void func() {
    char* env = getenv("LANGUAGE");
    if (env != NULL) {
        strncpy(env, "C", 1);
        printstr(env);
    }
}
```

In this example, the first argument of `strncpy` is the return value from a nonreentrant standard function `getenv`. The behavior can be undefined because `strncpy` modifies this argument.

Correction - Copy Return Value of `getenv` and Modify Copy

One possible solution is to copy the return value of `getenv` and pass the copy to the `strncpy` function.

```
#include <stdlib.h>
#include <string.h>
enum {
    SIZE20 = 20
};

void printstr(const char*);

void func() {
    char* env = getenv("LANGUAGE");
    if (env != NULL) {
        char env_cp[SIZE20];
        strncpy(env_cp, env, SIZE20);
        strncpy(env_cp, "C", 1);
        printstr(env_cp);
    }
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

ENV30-C

Introduced in R2019a

CERT C++: ENV31-C

Do not rely on an environment pointer following an operation that may invalidate it

Description

Rule Definition

Do not rely on an environment pointer following an operation that may invalidate it.

Polyspace Implementation

This checker checks for **Environment pointer invalidated by previous operation**.

Examples

Environment pointer invalidated by previous operation

Issue

Environment pointer invalidated by previous operation occurs when you use the third argument of *main()* in a hosted environment to access the environment after an operation modifies the environment. In a hosted environment, many C implementations support the nonstandard syntax:

```
main (int argc, char *argv[], char *envp[])
```

A call to a `setenv` or `putenv` family function modifies the environment pointed to by `*envp`.

Risk

When you modify the environment through a call to a `setenv` or `putenv` family function, the environment memory can potentially be reallocated. The hosted environment pointer is not updated and might point to an incorrect location. A call to this pointer can return unexpected results or cause an abnormal program termination.

Fix

Do not use the hosted environment pointer. Instead, use global external variable `environ` in Linux, `_environ` or `_wenviron` in Windows, or their equivalent. When you modify the environment, these variables are updated.

Example - Access Environment Through Pointer envp

```
#include <stdio.h>
#include <stdlib.h>

extern int check_arguments(int argc, char **argv, char **envp);
extern void use_envp(char **envp);

/* envp is from main function */
int func(char **envp)
{
    /* Call to setenv may cause environment
     *memory to be reallocated
     */
```

```
    if (setenv(("MY_NEW_VAR"),("new_value"),1) != 0)
    {
        /* Handle error */
        return -1;
    }
    /* envp not updated after call to setenv, and may
     *point to incorrect location.
     **/
    if (envp != ((void *)0)) {
        use_envp(envp);
/* No defect on second access to
*envp because defect already raised */
    }
    return 0;
}

void  main(int argc, char **argv, char **envp)
{
    if (check_arguments(argc, argv, envp))
    {
        (void)func(envp);
    }
}
```

In this example, `envp` is accessed inside `func()` after a call to `setenv` that can reallocate the environment memory. `envp` can point to an incorrect location because it is not updated after `setenv` modifies the environment. No defect is raised when `use_envp()` is called because the defect is already raised on the previous line of code.

Correction — Use Global External Variable `environ`

One possible correction is to access the environment by using a variable that is always updated after a call to `setenv`. For instance, in the following code, the pointer `envp` is still available from `main()`, but the environment is accessed in `func()` through the global external variable `environ`.

```
#include <stdio.h>
#include <stdlib.h>
extern char **environ;

extern int check_arguments(int argc, char **argv, char **envp);
extern void use_envp(char **envp);

int func(void)
{
    if (setenv(("MY_NEW_VAR"), ("new_value"),1) != 0) {
        /* Handle error */
        return -1;
    }
  /* Use global external variable environ
   *which is always updated after a call to setenv */

    if (environ != NULL) {
        use_envp(environ);
    }
    return 0;
}

void  main(int argc, char **argv, char **envp)
```

```
{
    if (check_arguments(argc, argv, envp))
    {
        (void)func();
    }
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

ENV31-C

Introduced in R2019a

CERT C++: ENV32-C

All exit handlers must return normally

Description

Rule Definition

All exit handlers must return normally.

Polyspace Implementation

This checker checks for **Abnormal termination of exit handler**.

Examples

Abnormal termination of exit handler

Issue

Abnormal termination of exit handler looks for registered exit handlers. Exit handlers are registered with specific functions such as `atexit`, (WinAPI) `_onexit`, or `at_quick_exit()`. If the exit handler calls a function that interrupts the program's expected termination sequence, Polyspace raises a defect. Some functions that can cause abnormal exits are `exit`, `abort`, `longjmp`, or (WinAPI) `_onexit`.

Risk

If your exit handler terminates your program, you can have undefined behavior. Abnormal program termination means other exit handlers are not invoked. These additional exit handlers may do additional clean up or other required termination steps.

Fix

In inside exit handlers, remove calls to functions that prevent the exit handler from terminating normally.

Example - Exit Handler With Call to `exit`

```
#include <stdlib.h>

volatile int some_condition = 1;
void demo_exit1(void)
{
    /* ... Cleanup code ... */
    return;
}
void exitabnormalhandler(void)
{
    if (some_condition)
    {
        /* Clean up */
        exit(0);
    }
    return;
```

```
}

int demo_install_exitabnormalhandler(void)
{

    if (atexit(demo_exit1) != 0) /* demo_exit1() performs additional cleanup */
    {
        /* Handle error */
    }
    if (atexit(exitabnormalhandler) != 0)
    {
        /* Handle error */
    }
    /* ... Program code ... */
    return 0;
}
```

In this example, `demo_install_exitabnormalhandler` registers two exit handlers, `demo_exit1` and `exitabnormalhandler`. Exit handlers are invoked in the reverse order of which they are registered. When the program ends, `exitabnormalhandler` runs, then `demo_exit1`. However, `exitabnormalhandler` calls `exit` interrupting the program exit process. Having this `exit` inside an exit handler causes undefined behavior because the program is not finished cleaning up safely.

Correction — Remove `exit` from Exit Handler

One possible correction is to let your exit handlers terminate normally. For this example, `exit` is removed from `exitabnormalhandler`, allowing the exit termination process to complete as expected.

```
#include <stdlib.h>

volatile int some_condition = 1;
void demo_exit1(void)
{
    /* ... Cleanup code ... */
    return;
}
void exitabnormalhandler(void)
{
    if (some_condition)
    {
        /* Clean up */
        /* Return normally */
    }
    return;
}

int demo_install_exitabnormalhandler(void)
{

    if (atexit(demo_exit1) != 0) /* demo_exit1() continues clean up */
    {
        /* Handle error */
    }
    if (atexit(exitabnormalhandler) != 0)
    {
        /* Handle error */
    }
```

```
    /* ... Program code ... */
    return 0;
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

ENV32-C

Introduced in R2019a

CERT C++: ENV33-C

Do not call system()

Description

Rule Definition

Do not call system().

Polyspace Implementation

This checker checks for **Unsafe call to a system function**.

Examples

Unsafe call to a system function

Issue

Unsafe call to a system function occurs when you use a function that invokes an implementation-defined command processor. These functions include:

- The C standard `system()` function.
- The POSIX `popen()` function.
- The Windows `_popen()` and `_wpopen()` functions.

Risk

If the argument of a function that invokes a command processor is not sanitized, it can cause exploitable vulnerabilities. An attacker can execute arbitrary commands or read and modify data anywhere on the system.

Fix

Do not use a `system`-family function to invoke a command processor. Instead, use safer functions such as POSIX `execve()` and WinAPI `CreateProcess()`.

Example - `system()` Called

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};

void func(char *arg)
{
    char buf[SIZE512];
    int retval=sprintf(buf, "/usr/bin/any_cmd %s", arg);
```

```
    if (retval<=0 || retval>SIZE512){
        /* Handle error */
        abort();
    }
    /* Use of system() to pass any_cmd with
    unsanitized argument to command processor */

    if (system(buf) == -1) {
    /* Handle error */
  }
}
```

In this example, `system()` passes its argument to the host environment for the command processor to execute. This code is vulnerable to an attack by command-injection.

Correction — Sanitize Argument and Use `execve()`

In the following code, the argument of `any_cmd` is sanitized, and then passed to `execve()` for execution. `exec`-family functions are not vulnerable to command-injection attacks.

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};


void func(char *arg)
{
  char *const args[SIZE3] = {"any_cmd", arg, NULL};
  char  *const env[] = {NULL};

  /* Sanitize argument */

  /* Use execve() to execute any_cmd. */

  if (execve("/usr/bin/time", args, env) == -1) {
    /* Handle error */
  }
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

ENV33-C

Introduced in R2019a

CERT C++: ENV34-C

Do not store pointers returned by certain functions

Description

Rule Definition

Do not store pointers returned by certain functions.

Polyspace Implementation

This checker checks for **Misuse of return value from nonreentrant standard function**.

Examples

Misuse of return value from nonreentrant standard function

Issue

Misuse of return value from nonreentrant standard function occurs when these events happen in this sequence:

1. You point to the buffer returned from a nonreentrant standard function such as `getenv` or `setlocale`.

   ```
   user = getenv("USER");
   ```

2. You call that nonreentrant standard function again.

   ```
   user2 = getenv("USER2");
   ```

3. You use or dereference the pointer from the first step expecting the buffer to remain unmodified since that step. In the meantime, the call in the second step has modified the buffer.

   For instance:

   ```
   var=*user;
   ```

In some cases, the defect might appear even if you do not call the `getenv` function a second time but simply return the pointer. For instance:

```
char* func() {
     user=getenv("USER");
     .
     .
     return user;
}
```

For information on which functions are covered by this defect, see documentation on nonreentrant standard functions.

Risk

The C Standard allows nonreentrant functions such as `getenv` to return a pointer to a *static* buffer. Because the buffer is static, a second call to `getenv` modifies the buffer. If you continue to use the

pointer returned from the first call past the second call, you can see unexpected results. The buffer that it points to no longer has values from the first call.

The defect appears even if you do not call `getenv` a second time but simply return the pointer. The reason is that someone calling your function might use the returned pointer *after* a second call to `getenv`. By returning the pointer from your call to `getenv`, you make your function unsafe to use.

The same rationale is true for other nonreentrant functions covered by this defect.

Fix

After the first call to `getenv`, make a copy of the buffer that the returned pointer points to. After the second call to `getenv`, use this copy. Even if the second call modifies the buffer, your copy is untouched.

Example - Return from getenv Used After Second Call to getenv

```
#include <stdlib.h>
#include <string.h>

int func()
{
    int result = 0;

    char *home = getenv("HOME");   /* First call */
    if (home != NULL) {
        char *user = NULL;
        char *user_name_from_home = strrchr(home, '/');

        if (user_name_from_home != NULL) {
            user = getenv("USER");   /* Second call */
            if ((user != NULL) &&
                (strcmp(user, user_name_from_home) == 0))
            {
                result = 1;
            }
        }
    }
    return result;
}
```

In this example, the pointer `user_name_from_home` is derived from the pointer `home`. `home` points to the buffer returned from the first call to `getenv`. Therefore, `user_name_from_home` points to a location in the same buffer.

After the second call to `getenv`, the buffer is modified. If you continue to use `user_name_from_home`, you can get unexpected results.

Correction — Make Copy of Buffer Before Second Call

If you want to access the buffer from the first call to `getenv` past the second call, make a copy of the buffer after the first call. One possible correction is to use the `strdup` function to make the copy.

```
#include <stdlib.h>
#include <string.h>

int func()
{
```

```
    int result = 0;

    char *home = getenv("HOME");
    if (home != NULL) {
        char *user = NULL;
        char *user_name_from_home = strrchr(home, '/');
        if (user_name_from_home != NULL) {
            /* Make copy before second call */
            char *saved_user_name_from_home = strdup(user_name_from_home);
            if (saved_user_name_from_home != NULL) {
                user = getenv("USER");
                if ((user != NULL) &&
                    (strcmp(user, saved_user_name_from_home) == 0))
                {
                    result = 1;
                }
                free(saved_user_name_from_home);
            }
        }
    }
    return result;
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

ENV34-C

Introduced in R2019a

CERT C++: FLP30-C

Do not use floating-point variables as loop counters

Description

Rule Definition

Do not use floating-point variables as loop counters.

Polyspace Implementation

This checker checks for **Floating type or multiple for loop counters**.

Examples

Floating type or multiple for loop counters

Issue

The checker flags these situations:

- The `for` loop index has a floating point type.
- More than one loop counter is incremented in the `for` loop increment statement.

  For instance:

  ```
  for(i=0, j=0; i<10 && j < 10;i++, j++) {}
  ```
- A loop counter is not incremented in the `for` loop increment statement.

  For instance:

  ```
  for(i=0; i<10;) {}
  ```

  Even if you increment the loop counter in the loop body, the checker still raises a violation.

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FLP30-C

Introduced in R2019a

CERT C++: FLP32-C

Prevent or detect domain and range errors in math functions

Description

Rule Definition

Prevent or detect domain and range errors in math functions.

Polyspace Implementation

This checker checks for **Invalid use of standard library floating point routine**.

Examples

Invalid use of standard library floating point routine

Issue

Invalid use of standard library floating point routine occurs when you use invalid arguments with a floating point function from the standard library. This defect picks up:

- Rounding and absolute value routines

  ```
  ceil, fabs, floor, fmod
  ```
- Fractions and division routines

  ```
  fmod, modf
  ```
- Exponents and log routines

  ```
  frexp, ldexp, sqrt, pow, exp, log, log10
  ```
- Trigonometry function routines

  ```
  cos, sin, tan, acos, asin, atan, atan2, cosh, sinh, tanh, acosh, asinh,
  atanh
  ```

Risk

Domain errors on standard library floating point functions result in implementation-defined values. If you use the function return value in subsequent computations, you can see unexpected results.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the function argument acquires invalid values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

It is a good practice to handle for domain errors before using a standard library floating point function. For instance, before calling the `acos` function, check if the argument is in [-1.0, 1.0] and handle the error.

See examples of fixes below.

If you do not want to fix the issue, for instance, when you handle infinities in your code, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Arc Cosine Operation

```
#include <math.h>

double arccosine(void) {
    double degree = 5.0;
    return acos(degree);
}
```

The input value to `acos` must be in the interval `[-1,1]`. This input argument, `degree`, is outside this range.

Correction — Change Input Argument

One possible correction is to change the input value to fit the specified range. In this example, change the input value from degrees to radians to fix this defect.

```
#include <math.h>

double arccosine(void) {
    double degree = 5.0;
    double radian = degree * 3.14159 / 180.;
    return acos(radian);
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FLP32-C

Introduced in R2019a

CERT C++: FLP34-C

Ensure that floating-point conversions are within range of the new type

Description

Rule Definition

Ensure that floating-point conversions are within range of the new type.

Polyspace Implementation

This checker checks for **Float conversion overflow**.

Examples

Float conversion overflow

Issue

Float conversion overflow occurs when converting a floating point number to a smaller floating point data type. If the variable does not have enough memory to represent the original number, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

Risk

Overflows can result in unpredictable values from computations. The result can be infinity or the maximum finite value depending on the rounding mode used in the implementation. If you use the result of an overflowing conversion in subsequent computations and do not account for the overflow, you can see unexpected results.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variable being converted acquires its current value You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller floating point types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Converting from `double` to `float`

```
float convert(void) {

    double diam = 1e100;
    return (float)diam;
}
```

In the return statement, the variable `diam` of type double (64 bits) is converted to a variable of type float (32 bits). However, the value 1^100 requires more than 32 bits to be precisely represented.

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FLP34-C

Introduced in R2019a

CERT C++: FLP36-C

Preserve precision when converting integral values to floating-point type

Description

Rule Definition

Preserve precision when converting integral values to floating-point type.

Polyspace Implementation

This checker checks for **Precision loss in integer to float conversion**.

Examples

Precision loss in integer to float conversion

Issue

Precision loss from integer to float conversion occurs when you cast an integer value to a floating-point type that cannot represent the original integer value.

For instance, the `long int` value `1234567890L` is too large for a variable of type `float` .

Risk

If the floating-point type cannot represent the integer value, the behavior is undefined (see C11 standard, 6.3.1.4, paragraph 2). For instance, least significant bits of the variable value can be dropped leading to unexpected results.

Fix

Convert to a floating-point type that can represent the integer value.

For instance, if the `float` data type cannot represent the integer value, use the `double` data type instead.

When writing a function that converts an integer to floating point type, before the conversion, check if the integer value can be represented in the floating-point type. For instance, `DBL_MANT_DIG * log2(FLT_RADIX)` represents the number of base-2 digits in the type `double`. Before conversion to the type `double`, check if this number is greater than or equal to the precision of the integer that you are converting. To determine the precision of an integer `num`, use this code:

```
size_t precision = 0;
while (num != 0) {
   if (num % 2 == 1) {
     precision++;
   }
   num >>= 1;
}
```

Some implementations provide a builtin function to determine the precision of an integer. For instance, GCC provides the function `__builtin_popcount`.

Example - Conversion of Large Integer to Floating-Point Type

```
#include <stdio.h>

int main(void) {
  long int big = 1234567890L;
  float approx = big;
  printf("%ld\n", (big - (long int)approx));
  return 0;
}
```

In this example, the `long int` variable `big` is converted to `float`.

Correction — Use a Wider Floating-Point Type

One possible correction is to convert to the `double` data type instead of `float`.

```
#include <stdio.h>

int main(void) {
  long int big = 1234567890L;
  double approx = big;
  printf("%ld\n", (big - (long int)approx));
  return 0;
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FLP36-C

Introduced in R2019a

CERT C++: FLP37-C

Do not use object representations to compare floating-point values

Description

Rule Definition

Do not use object representations to compare floating-point values.

Polyspace Implementation

This checker checks for **Memory comparison of float-point values**.

Examples

Memory comparison of float-point values

Issue

Memory comparison of float-point values occurs when you compare the object representation of floating-point values or the object representation of structures containing floating-point members. When you use the functions `memcmp`, `bcmp`, or `wmemcmp` to perform the bit pattern comparison, the defect is raised.

Risk

The object representation of floating-point values uses specific bit patterns to encode those values. Floating-point values that are equal, for instance `-0.0` and `0.0` in the IEC 60559 standard, can have different bit patterns in their object representation. Similarly, floating-point values that are not equal can have the same bit pattern in their object representation.

Fix

When you compare structures containing floating-point members, compare the structure members individually.

To compare two floating-point values, use the `==` or `!=` operators. If you follow a standard that discourages the use of these operators, such as MISRA, ensure that the difference between the floating-point values is within an acceptable range.

Example - Using `memcmp` to Compare Structures with Floating-Point Members

```
#include <string.h>

typedef struct {
    int i;
    float f;
} myStruct;

extern void initialize_Struct(myStruct *);

int func_cmp(myStruct *s1, myStruct *s2) {
/* Comparison between structures containing
```

```
* floating-point members */
    return memcmp
        ((const void *)s1, (const void *)s2, sizeof(myStruct));
}

void func(void) {
    myStruct s1, s2;
    initialize_Struct(&s1);
    initialize_Struct(&s2);
    (void)func_cmp(&s1, &s2);
}
```

In this example, `func_cmp()` calls `memcmp()` to compare the object representations of structures `s1` and `s2`. The comparison might be inaccurate because the structures contain floating-point members.

Correction — Compare Structure Members Individually

One possible correction is to compare the structure members individually and to ensure that the difference between the floating-point values is within an acceptable range defined by `ESP`.

```
 #include <string.h>

typedef struct {
    int i;
    float f;
} myStruct;

extern void initialize_Struct(myStruct *);

#define ESP 0.00001

int func_cmp(myStruct *s1, myStruct *s2) {

/*Structure members are compared individually */
    return ((s1->i == s2->i) &&
            (fabsf(s1->f - s2->f) <= ESP));
}

void func(void) {
    myStruct s1, s2;
    initialize_Struct(&s1);
    initialize_Struct(&s2);
    (void)func_cmp(&s1, &s2);
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

FLP37-C

Introduced in R2019a

CERT C++: MSC30-C

Do not use the rand() function for generating pseudorandom numbers

Description

Rule Definition

Do not use the rand() function for generating pseudorandom numbers.

Polyspace Implementation

This checker checks for **Vulnerable pseudo-random number generator**.

Examples

Vulnerable pseudo-random number generator

Issue

The **Vulnerable pseudo-random number generator** identifies uses of cryptographically weak pseudo-random number generator (PRNG) routines.

The list of cryptographically weak routines flagged by this checker include:

- `rand`, `random`
- `drand48`, `lrand48`, `mrand48`, `erand48`, `nrand48`, `jrand48`, and their `_r` equivalents such as `drand48_r`
- `RAND_pseudo_bytes`

Risk

These cryptographically weak routines are predictable and must not be used for security purposes. When a predictable random value controls the execution flow, your program is vulnerable to malicious attacks.

Fix

Use more cryptographically sound random number generators, such as `CryptGenRandom` (Windows), `OpenSSL/RAND_bytes`(Linux/UNIX).

Example - Random Loop Numbers

```
#include <stdio.h>
#include <stdlib.h>

volatile int rd = 1;
int main(int argc, char *argv[])
{
    int j, r, nloops;
    struct random_data buf;
    int i = 0;
```

```
    nloops = rand();

    for (j = 0; j < nloops; j++) {
        if (random_r(&buf, &i))
            exit(1);
        printf("random_r: %ld\n", (long)i);
    }
    return 0;
}
```

This example uses `rand` and `random_r` to generate random numbers. If you use these functions for security purposes, these PRNGs can be the source of malicious attacks.

Correction — Use Stronger PRNG

One possible correction is to replace the vulnerable PRNG with a stronger random number generator.

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/rand.h>

volatile int rd = 1;
int main(int argc, char* argv[])
{
    int j, r, nloops;
    unsigned char buf;
    unsigned int seed;
    int i = 0;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <seed> <nloops>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    seed = atoi(argv[1]);
    nloops = atoi(argv[2]);

    for (j = 0; j < nloops; j++) {
        if (RAND_bytes(&buf, i) != 1)
            exit(1);
        printf("RAND_bytes: %u\n", (unsigned)buf);
    }
    return 0;
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites
MSC30-C

Introduced in R2019a

CERT C++: MSC32-C

Properly seed pseudorandom number generators

Description

Rule Definition

Properly seed pseudorandom number generators.

Polyspace Implementation

This checker checks for these issues:

- **Deterministic random output from constant seed**.
- **Predictable random output from predictable seed**.

Examples

Deterministic random output from constant seed

Issue

Deterministic random output from constant seed detects random standard functions that when given a constant seed, have deterministic output.

Risk

When some random functions, such as `srand`, `srandom`, and `initstate`, have constant seeds, the results produce the same output every time that your program is run. A hacker can disrupt your program if they know how your program behaves.

Fix

Use a different random standard function or use a nonconstant seed.

Some standard random routines are inherently cryptographically weak on page 3-405, and should not be used for security purposes.

Example - Random Number Generator Initialization

```
#include <stdlib.h>

void random_num(void)
{
    srand(12345U);
    /* ... */
}
```

This example initializes a random number generator using `srand` with a constant seed. The random number generation is deterministic, making this function cryptographically weak.

Correction — Use Different Random Number Generator

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S
#include <stdlib.h>
#include <stdio.h>

unsigned int random_num_time(void)
{

    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

Predictable random output from predictable seed

Issue

Predictable random output from predictable seed looks for random standard functions that use a nonconstant but predictable seed. Examples of predictable seed generators are `time`, `gettimeofday`, and `getpid`.

Risk

When you use predictable seed values for random number generation, your random numbers are also predictable. A hacker can disrupt your program if they know how your program behaves.

Fix

You can use a different function to generate less predictable seeds.

You can also use a different random number generator that does not require a seed. For example, the Windows API function `rand_s` seeds itself by default. It uses information from the entire system, for example, system time, thread ids, system counter, and memory clusters. This information is more random and a user cannot access this information.

Some standard random routines are inherently cryptographically weak on page 3-405, and should not be used for security purposes.

Example - Seed as an Argument

```
#include <stdlib.h>
#include <time.h>

void seed_rng(int seed)
```

```
{
    srand(seed);
}

int generate_num(void)
{
    seed_rng(time(NULL) + 3);
    /* ... */
}
```

This example uses `srand` to start the random number generator with `seed` as the seed. However, `seed` is predictable because the function `time` generates it. So, an attacker can predict the random numbers generated by `srand`.

Correction — Use Different Random Number Generator

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int generate_num(void)
{
    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MSC32-C

Introduced in R2019a

CERT C++: MSC33-C

Do not pass invalid data to the asctime() function

Description

Rule Definition

Do not pass invalid data to the asctime() function.

Polyspace Implementation

This checker checks for **Use of obsolete standard function**.

Examples

Use of obsolete standard function

Issue

Use of obsolete standard function detects calls to standard function routines that are considered legacy, removed, deprecated, or obsolete by C/C++ coding standards.

Obsolete Function	Standards	Risk	Replacement Function
`asctime`	Deprecated in POSIX.1-2008	Not thread-safe.	`strftime` or `asctime_s`
`asctime_r`	Deprecated in POSIX.1-2008	Implementation based on unsafe function `sprintf`.	`strftime` or `asctime_s`
`bcmp`	Deprecated in 4.3BSD Marked as legacy in POSIX.1-2001.	Returns from function after finding the first differing byte, making it vulnerable to timing attacks.	`memcmp`
`bcopy`	Deprecated in 4.3BSD Marked as legacy in POSIX.1-2001.	Returns from function after finding the first differing byte, making it vulnerable to timing attacks.	`memcpy` or `memmove`
`brk` and `sbrk`	Marked as legacy in SUSv2 and POSIX.1-2001.		`malloc`
`bsd_signal`	Removed in POSIX.1-2008		`sigaction`
`bzero`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`memset`
`ctime`	Deprecated in POSIX.1-2008	Not thread-safe.	`strftime` or `asctime_s`

Obsolete Function	Standards	Risk	Replacement Function
`ctime_r`	Deprecated in POSIX.1-2008	Implementation based on unsafe function `sprintf`.	`strftime` or `asctime_s`
`cuserid`	Removed in POSIX.1-2001.	Not reentrant. Precise functionality not standardized causing portability issues.	`getpwuid`
`ecvt` and `fcvt`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008	Not reentrant	`snprintf`
`ecvt_r` and `fcvt_r`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008		`snprintf`
`ftime`	Removed in POSIX.1-2008		`time`, `gettimeofday`, `clock_gettime`
`gamma`, `gammaf`, `gammal`	Function not specified in any standard because of historical variations	Portability issues.	`tgamma`, `lgamma`
`gcvt`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`snprintf`
`getcontext`	Removed in POSIX.1-2008.	Portability issues.	Use POSIX thread instead.
`getdtablesize`	BSD API function not included in POSIX.1-2001	Portability issues.	`sysconf( _SC_OPEN_MAX )`
`gethostbyaddr`	Removed in POSIX.1-2008	Not reentrant	`getaddrinfo`
`gethostbyname`	Removed in POSIX.1-2008	Not reentrant	`getnameinfo`
`getpagesize`	BSD API function not included in POSIX.1-2001	Portability issues.	`sysconf( _SC_PAGESIZE )`
`getpass`	Removed in POSIX.1-2001.	Not reentrant.	`getpwuid`
`getw`	Not present in POSIX.1-2001.		`fread`
`getwd`	Marked legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`getcwd`
`index`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`strchr`
`makecontext`	Removed in POSIX.1-2008.	Portability issues.	Use POSIX thread instead.
`memalign`	Appears in SunOS 4.1.3. Not in 4.4 BSD or POSIX.1-2001		`posix_memalign`
`mktemp`	Removed in POSIX.1-2008.	Generated names are predictable and can cause a race condition.	`mkstemp` removes race risk

Obsolete Function	Standards	Risk	Replacement Function
`pthread_attr_getstackaddr` and `pthread_attr_setstackaddr`		Ambiguities in the specification of the `stackaddr` attribute cause portability issues	`pthread_attr_getstack` and `pthread_attr_setstack`
`putw`	Not present in POSIX.1-2001.	Portability issues.	`fwrite`
`qecvt` and `qfcvt`	Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008		`snprintf`
`qecvt_r` and `qfcvt_r`	Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008		`snprintf`
`rand_r`	Marked as obsolete in POSIX.1-2008		
`re_comp`	BSD API function	Portability issues	`regcomp`
`re_exes`	BSD API function	Portability issues	`regexec`
`rindex`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.		`strrchr`
`scalb`	Removed in POSIX.1-2008		`scalbln`, `scalblnf`, or `scalblnl`
`sigblock`	4.3BSD signal API whose origin is unclear		`sigprocmask`
`sigmask`	4.3BSD signal API whose origin is unclear		`sigprocmask`
`sigsetmask`	4.3BSD signal API whose origin is unclear		`sigprocmask`
`sigstack`	Interface is obsolete and not implemented on most platforms.	Portability issues.	`sigaltstack`
`sigvec`	4.3BSD signal API whose origin is unclear		`sigaction`
`swapcontext`	Removed in POSIX.1-2008	Portability issues.	Use POSIX threads.
`tmpnam` and `tmpnam_r`	Marked as obsolete in POSIX.1-2008.	This function generates a different string each time it is called, up to TMP_MAX times. If it is called more than TMP_MAX times, the behavior is implementation-defined.	`mkstemp`, `tmpfile`
`ttyslot`	Removed in POSIX.1-2001.		
`ualarm`	Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008.	Errors are under-specified	`setitimer` or POSIX `timer_create`
`usleep`	Removed in POSIX.1-2008.		`nanosleep`
`utime`	SVr4, POSIX.1-2001. POSIX.1-2008 marks as obsolete.		

Obsolete Function	Standards	Risk	Replacement Function
`valloc`	Marked as obsolete in 4.3BSD. Marked as legacy in SUSv2. Removed from POSIX.1-2001		`posix_memalign`
`vfork`	Removed from POSIX.1-2008	Under-specified in previous standards.	`fork`
`wcswcs`	This function was not included in the final ISO/IEC 9899:1990/ Amendment 1:1995 (E).		`wcsstr`
`WinExec`	WinAPI provides this function only for 16-bit Windows compatibility.		`CreateProcess`
`LoadModule`	WinAPI provides this function only for 16-bit Windows compatibility.		`CreateProcess`

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Printing Out Time

```
#include <stdio.h>
#include <time.h>

void timecheck_bad(int argc, char *argv[])
{
    time_t ticks;

    ticks = time(NULL);
    printf("%.24s\r\n", ctime(&ticks));
}
```

In this example, the function `ctime` formats the current time and prints it out. However, `ctime` was removed after C99 because it does not work on multithreaded programs.

Correction — Different Time Function

One possible correction is to use `strftime` instead because this function uses a set buffer size.

```
#include <stdio.h>
#include <string.h>
#include <time.h>

void timecheck_good(int argc, char *argv[])
{
```

```
    char outBuff[1025];
    time_t ticks;
    struct tm * timeinfo;

    memset(outBuff, 0, sizeof(outBuff));

    ticks = time(NULL);
    timeinfo = localtime(&ticks);
    strftime(outBuff,sizeof(outBuff),"%I:%M%p.",timeinfo);
    fprintf(stdout, outBuff);
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MSC33-C

Introduced in R2019a

CERT C++: MSC37-C

Ensure that control never reaches the end of a non-void function

Description

Rule Definition

Ensure that control never reaches the end of a non-void function.

Polyspace Implementation

This checker checks for **Missing return statement**.

Examples

Missing return statement

Issue

Missing return statement occurs when a function does not return a value along at least one execution path. If the return type of the function is `void`, this error does not occur.

Risk

If a function has a non-`void` return value in its signature, it is expected to return a value. The return value of this function can be used in later computations. If the execution of the function body goes through a path where a `return` statement is missing, the function return value is indeterminate. Computations with this return value can lead to unpredictable results.

Fix

In most cases, you can fix this defect by placing the `return` statement at the end of the function body.

Alternatively, you can identify which execution paths through the function body do not have a `return` statement and add a `return` statement on those paths. Often the result details show a sequence of events that indicate this execution path. You can add a `return` statement at an appropriate point in the path. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Missing or invalid return statement error

```
int AddSquares(int n)
 {
   int i=0;
   int sum=0;
```

```
   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }
 }
/* Defect: No return value if n is not 0*/
```

If `n` is equal to 0, the code does not enter the `if` statement. Therefore, the function `AddSquares` does not return a value if `n` is 0.

Correction — Place Return Statement on Every Execution Path

One possible correction is to return a value in every branch of the `if...else` statement.

```
 int AddSquares(int n)
 {
   int i=0;
   int sum=0;

   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }

   /*Fix: Place a return statement on branches of if-else */
   else
     return 0;
  }
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MSC37-C

Introduced in R2019a

CERT C++: MSC38-C

Do not treat a predefined identifier as an object if it might only be implemented as a macro

Description

Rule Definition

Do not treat a predefined identifier as an object if it might only be implemented as a macro.

Polyspace Implementation

This checker checks for **Predefined macro used as an object**.

Examples

Predefined macro used as an object

Issue

Predefined macro used as an object occurs when you use certain identifiers in a way that requires an underlying object to be present. These identifiers are defined as macros. The C Standard does not allow you to redefine them as objects. You use the identifiers in such a way that macro expansion of the identifiers cannot occur.

For instance, you refer to an external variable `errno`:

```
extern int errno;
```

However, `errno` does not occur as a variable but a macro.

The defect applies to these macros: `assert`, `errno`, `math_errhandling`, `setjmp`, `va_arg`, `va_copy`, `va_end`, and `va_start`. The checker looks for the defect only in source files (not header files).

Risk

The C11 Standard (Sec. 7.1.4) allows you to redefine most macros as objects. To access the object and not the macro in a source file, you do one of these:

- Redeclare the identifier as an external variable or function.
- For function-like macros, enclose the identifier name in parentheses.

If you try to use these strategies for macros that cannot be redefined as objects, an error occurs.

Fix

Do not use the identifiers in such a way that a macro expansion is suppressed.

- Do not redeclare the identifiers as external variables or functions.
- For function-like macros, do not enclose the macro name in parentheses.

Example - Use of `assert` as Function

```
#include<assert.h>
typedef void (*err_handler_func)(int);

extern void demo_handle_err(err_handler_func, int);

void func(int err_code) {
    extern void assert(int);
    demo_handle_err(&(assert), err_code);
}
```

In this example, the `assert` macro is redefined as an external function. When passed as an argument to `demo_handle_err`, the identifier `assert` is enclosed in parentheses, which suppresses use of the `assert` macro.

Correction — Use `assert` as Macro

One possible correction is to directly use the `assert` macro from `assert.h`. A different implementation of the function `demo_handle_err` directly uses the `assert` macro instead of taking the address of an `assert` function.

```
#include<assert.h>
void demo_handle_err(int err_code) {
    assert(err_code == 0);
}

void func(int err_code) {
    demo_handle_err(err_code);
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MSC38-C

Introduced in R2019a

CERT C++: MSC39-C

Do not call va_arg() on a va_list that has an indeterminate value

Description

Rule Definition

Do not call va_arg() on a va_list that has an indeterminate value.

Polyspace Implementation

This checker checks for these issues:

- **Invalid va_list argument**.
- **Too many va_arg calls for current argument list**.

Examples

Invalid va_list argument

Issue

Invalid va_list argument occurs when you use a `va_list` variable as an argument to a function in the `vprintf` group but:

- You do not initialize the variable previously using `va_start` or `va_copy`.
- You invalidate the variable previously using `va_end` and do not reinitialize it.

For instance, you call the function `vsprintf` as `vsprintf (buffer,format, args)`. However, before the function call, you do not initialize the `va_list` variable `args` using either of the following:

- `va_start(args, paramName)`. `paramName` is the last named argument of a variable-argument function. For instance, for the function definition `void func(int n, char c, ...) {}`, `c` is the last named argument.
- `va_copy(args, anotherList)`. `anotherList` is another valid `va_list` variable.

Risk

The behavior of an uninitialized `va_list` argument is undefined. Calling a function with an uninitialized `va_list` argument can cause stack overflows.

Fix

Before using a `va_list` variable as function argument, initialize it with `va_start` or `va_copy`.

Clean up the variable using `va_end` only after all uses of the variable.

Example - `va_list` Variable Used Following Call to `va_end`

```
#include <stdarg.h>
#include <stdio.h>
```

```
int call_vfprintf(int line, const char *format, ...) {
    va_list ap;
    int r=0;

    va_start(ap, format);
    r = vfprintf(stderr, format, ap);
    va_end(ap);

    r += vfprintf(stderr, format, ap);
    return r;
}
```

In this example, the `va_list` variable `ap` is used in the `vfprintf` function, after the `va_end` macro is called.

Correction — Call `va_end` After Using `va_list` Variable

One possible correction is to call `va_end` only after all uses of the `va_list` variable.

```
#include <stdarg.h>
#include <stdio.h>

int call_vfprintf(int line, const char *format, ...) {
    va_list ap;
    int r=0;

    va_start(ap, format);
    r = vfprintf(stderr, format, ap);
    r += vfprintf(stderr, format, ap);
    va_end(ap);

    return r;
}
```

Too many va_arg calls for current argument list

Issue

Too many va_arg calls for current argument list occurs when the number of calls to `va_arg` exceeds the number of arguments passed to the corresponding variadic function. The analysis raises a defect only when the variadic function is called.

Too many va_arg calls for current argument list does not raise a defect when:

- The number of calls to `va_arg` inside the variadic function is indeterminate. For example, if the calls are from an external source.
- The `va_list` used in `va_arg` is invalid.

Risk

When you call `va_arg` and there is no next argument available in `va_list`, the behavior is undefined. The call to `va_arg` might corrupt data or return an unexpected result.

Fix

Ensure that you pass the correct number of arguments to the variadic function.

Example - No Argument Available When Calling va_arg

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/
int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
        if (count > 0) {
/* No further argument available
* in va_list when calling va_arg
*/

            result += va_arg(ap, int);
        }
    }
    va_end(ap);
    return result;
}

void func(void) {

    (void)variadic_func(2, 100);

}
```

In this example, the named argument and only one variadic argument are passed to `variadic_func()` when it is called inside `func()`. On the second call to `va_arg`, no further variadic argument is available in `ap` and the behavior is undefined.

Correction — Pass Correct Number of Arguments to Variadic Function

One possible correction is to ensure that you pass the correct number of arguments to the variadic function.

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/

int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
```

```
        if (count > 0) {

/* The correct number of arguments is
* passed to va_list when variadic_func()
* is called inside func()
*/
            result += va_arg(ap, int);
        }
    }
    va_end(ap);
    return result;
}

void func(void) {

    (void)variadic_func(2, 100, 200);

}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MSC39-C

Introduced in R2019a

CERT C++: MSC40-C

Do not violate constraints

Description

Rule Definition

Do not violate constraints.

Polyspace Implementation

This checker checks for **Inline constraint not respected**.

Examples

Inline constraint not respected

Issue

Inline constraint not respected occurs when you refer to a file scope modifiable static variable or define a local modifiable static variable in a nonstatic inlined function. The checker considers a variable as modifiable if it is not `const`-qualified.

For instance, `var` is a modifiable `static` variable defined in an `inline` function `func`. `g_step` is a file scope modifiable static variable referred to in the same inlined function.

```
static int g_step;
inline void func (void) {
   static int var = 0;
   var += g_step;
}
```

Risk

When you modify a static variable in multiple function calls, you expect to modify the same variable in each call. For instance, each time you call `func`, the same instance of `var1` is incremented but a separate instance of `var2` is incremented.

```
void func(void) {
   static var1 = 0;
   var2 = 0;
   var1++;
   var2++;
}
```

If a function has an inlined and non-inlined definition (in separate files), when you call the function, the C standard allows compilers to use either the inlined or the non-inlined form (see ISO/IEC 9899:2011, sec. 6.7.4). If your compiler uses an inlined definition in one call and the non-inlined definition in another, you are no longer modifying the same variable in both calls. This behavior defies the expectations from a static variable.

Fix

Use one of these fixes:

- If you do not intend to modify the variable, declare it as `const`.

  If you do not modify the variable, there is no question of unexpected modification.
- Make the variable non-`static`. Remove the `static` qualifier from the declaration.

  If the variable is defined in the function, it becomes a regular local variable. If defined at file scope, it becomes an extern variable. Make sure that this change in behavior is what you intend.
- Make the function `static`. Add a `static` qualifier to the function definition.

  If you make the function `static`, the file with the inlined definition always uses the inlined definition when the function is called. Other files use another definition of the function. The question of which function definition gets used is not left to the compiler.

Example - Static Variable Use in Inlined and External Definition

```
/* file1. c  : contains inline definition of get_random()*/

inline unsigned int get_random(void)
{

    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}


int call_get_random(void)
{
    unsigned int rand_no;
    int ii;
    for (ii = 0; ii < 100; ii++) {
         rand_no = get_random();
    }
    rand_no = get_random();
    return 0;
}

/* file2. c  : contains external definition of get_random()*/

extern unsigned int get_random(void)
{
    /* Initialize seeds */
    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}
```

In this example, `get_random()` has an inline definition in `file1.c` and an external definition in `file2.c`. When `get_random` is called in `file1.c`, compilers are free to choose whether to use the inline or the external definition.

Depending on the definition used, you might or might not modify the version of `m_z` and `m_w` in the inlined version of `get_random()`. This behavior contradicts the usual expectations from a static variable. When you call `get_random()`, you expect to always modify the same `m_z` and `m_w`.

Correction — Make Inlined Function Static

One possible correction is to make the inlined `get_random()` static. Irrespective of your compiler, calls to `get_random()` in `file1.c` then use the inlined definition. Calls to `get_random()` in other files use the external definition. This fix removes the ambiguity about which definition is used and whether the static variables in that definition are modified.

```
/* file1. c  : contains inline definition of get_random()*/

static inline unsigned int get_random(void)
{

    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}


int call_get_random(void)
{
    unsigned int rand_no;
    int ii;
    for (ii = 0; ii < 100; ii++) {
         rand_no = get_random();
    }
    rand_no = get_random();
    return 0;
}

/* file2. c  : contains external definition of get_random()*/

extern unsigned int get_random(void)
{
    /* Initialize seeds */
    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MSC40-C

Introduced in R2019a

CERT C++: MSC41-C

Never hard code sensitive information

Description

Rule Definition

Never hard code sensitive information.

Polyspace Implementation

This checker checks for **Hard coded sensitive data**.

Examples

Hard coded sensitive data

Hard coded sensitive data occurs when data that is potentially sensitive is directly exposed in the code, for instance, as string literals. The checker identifies data as sensitive from their use in certain functions such as password encryption functions.

Following data can be potentially sensitive.

Type of Data	Functions That Indicate Sensitive Nature of Information
Host name	• `sethostname`, `setdomainname`, `gethostbyname`, `gethostbyname2`, `getaddrinfo`, `gethostbyname_r`, `gethostbyname2_r` (string argument) • `inet_aton`, `inet_pton`, `inet_net_pton`, `inet_addr`, `inet_network` (string argument) • `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (2nd argument)
Password	• `CreateProcessWithLogonW`, `LogonUser` (1st argument) • `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (3rd argument)

Type of Data	Functions That Indicate Sensitive Nature of Information
Database	• MySQL: `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (4th argument) • SQLite: `sqlite3_open`, `sqlite3_open16`, `sqlite3_open_v2` (1st argument) • PostgreSQL: `PQconnectdb` • Microsoft SQL: `SQLDriverConnect` (3rd argument)
User name	• `getpw`, `getpwnam`, `getpwnam_r`, `getpwuid`, `getpwuid_r`
Salt	`crypt`, `crypt_r` (2nd argument)
Cryptography keys and initialization vectors	OpenSSL: • `EVP_CipherInit`, `EVP_EncryptInit`, `EVP_DecryptInit` (3rd argument) • `EVP_CipherInit_ex`, `EVP_EncryptInit_ex`, `EVP_DecryptInit_ex` (4th argument)
Seed	• `srand`, `srandom`, `initstate` (1st argument) • OpenSSL: `RAND_seed`, `RAND_add`

Risk

Information that is hardcoded can be queried from binaries generated from the code.

Fix

Avoid hard coding sensitive information.

Check Information

Group: Rule 48. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MSC41-C

Introduced in R2020a

CERT C++: MSC50-CPP

Do not use std::rand() for generating pseudorandom numbers

Description

Rule Definition

Do not use std::rand() for generating pseudorandom numbers.

Polyspace Implementation

This checker checks for **Vulnerable pseudo-random number generator**.

Examples

Vulnerable pseudo-random number generator

Issue

The **Vulnerable pseudo-random number generator** identifies uses of cryptographically weak pseudo-random number generator (PRNG) routines.

The list of cryptographically weak routines flagged by this checker include:

- `rand`, `random`
- `drand48`, `lrand48`, `mrand48`, `erand48`, `nrand48`, `jrand48`, and their `_r` equivalents such as `drand48_r`
- `RAND_pseudo_bytes`

Risk

These cryptographically weak routines are predictable and must not be used for security purposes. When a predictable random value controls the execution flow, your program is vulnerable to malicious attacks.

Fix

Use more cryptographically sound random number generators, such as `CryptGenRandom` (Windows), `OpenSSL/RAND_bytes`(Linux/UNIX).

Example - Random Loop Numbers

```
#include <stdio.h>
#include <stdlib.h>

volatile int rd = 1;
int main(int argc, char *argv[])
{
    int j, r, nloops;
    struct random_data buf;
    int i = 0;
```

```
    nloops = rand();

    for (j = 0; j < nloops; j++) {
        if (random_r(&buf, &i))
            exit(1);
        printf("random_r: %ld\n", (long)i);
    }
    return 0;
}
```

This example uses `rand` and `random_r` to generate random numbers. If you use these functions for security purposes, these PRNGs can be the source of malicious attacks.

Correction — Use Stronger PRNG

One possible correction is to replace the vulnerable PRNG with a stronger random number generator.

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/rand.h>

volatile int rd = 1;
int main(int argc, char* argv[])
{
    int j, r, nloops;
    unsigned char buf;
    unsigned int seed;
    int i = 0;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <seed> <nloops>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    seed = atoi(argv[1]);
    nloops = atoi(argv[2]);

    for (j = 0; j < nloops; j++) {
        if (RAND_bytes(&buf, i) != 1)
            exit(1);
        printf("RAND_bytes: %u\n", (unsigned)buf);
    }
    return 0;
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites
MSC50-CPP

Introduced in R2019a

CERT C++: MSC51-CPP

Ensure your random number generator is properly seeded

Description

Rule Definition

Ensure your random number generator is properly seeded.

Polyspace Implementation

This checker checks for these issues:

- **Deterministic random output from constant seed**.
- **Predictable random output from predictable seed**.

Examples

Deterministic random output from constant seed

Issue

Deterministic random output from constant seed detects random standard functions that when given a constant seed, have deterministic output.

Risk

When some random functions, such as `srand`, `srandom`, and `initstate`, have constant seeds, the results produce the same output every time that your program is run. A hacker can disrupt your program if they know how your program behaves.

Fix

Use a different random standard function or use a nonconstant seed.

Some standard random routines are inherently cryptographically weak on page 3-405, and should not be used for security purposes.

Example - Random Number Generator Initialization

```
#include <stdlib.h>

void random_num(void)
{
    srand(12345U);
    /* ... */
}
```

This example initializes a random number generator using `srand` with a constant seed. The random number generation is deterministic, making this function cryptographically weak.

Correction — Use Different Random Number Generator

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S
#include <stdlib.h>
#include <stdio.h>

unsigned int random_num_time(void)
{

    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

Predictable random output from predictable seed

Issue

Predictable random output from predictable seed looks for random standard functions that use a nonconstant but predictable seed. Examples of predictable seed generators are `time`, `gettimeofday`, and `getpid`.

Risk

When you use predictable seed values for random number generation, your random numbers are also predictable. A hacker can disrupt your program if they know how your program behaves.

Fix

You can use a different function to generate less predictable seeds.

You can also use a different random number generator that does not require a seed. For example, the Windows API function `rand_s` seeds itself by default. It uses information from the entire system, for example, system time, thread ids, system counter, and memory clusters. This information is more random and a user cannot access this information.

Some standard random routines are inherently cryptographically weak on page 3-405, and should not be used for security purposes.

Example - Seed as an Argument

```
#include <stdlib.h>
#include <time.h>

void seed_rng(int seed)
```

```
{
    srand(seed);
}

int generate_num(void)
{
    seed_rng(time(NULL) + 3);
    /* ... */
}
```

This example uses `srand` to start the random number generator with `seed` as the seed. However, `seed` is predictable because the function `time` generates it. So, an attacker can predict the random numbers generated by `srand`.

Correction — Use Different Random Number Generator

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int generate_num(void)
{
    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MSC51-CPP

Introduced in R2019a

CERT C++: MSC52-CPP

Value-returning functions must return a value from all exit paths

Description

Rule Definition

Value-returning functions must return a value from all exit paths.

Polyspace Implementation

This checker checks for **Missing return statement**.

Examples

Missing return statement

Issue

Missing return statement occurs when a function does not return a value along at least one execution path. If the return type of the function is `void`, this error does not occur.

Risk

If a function has a non-`void` return value in its signature, it is expected to return a value. The return value of this function can be used in later computations. If the execution of the function body goes through a path where a `return` statement is missing, the function return value is indeterminate. Computations with this return value can lead to unpredictable results.

Fix

In most cases, you can fix this defect by placing the `return` statement at the end of the function body.

Alternatively, you can identify which execution paths through the function body do not have a `return` statement and add a `return` statement on those paths. Often the result details show a sequence of events that indicate this execution path. You can add a `return` statement at an appropriate point in the path. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Missing or invalid return statement error

```
int AddSquares(int n)
 {
   int i=0;
   int sum=0;
```

```
   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }
 }
/* Defect: No return value if n is not 0*/
```

If `n` is equal to 0, the code does not enter the `if` statement. Therefore, the function `AddSquares` does not return a value if `n` is 0.

Correction — Place Return Statement on Every Execution Path

One possible correction is to return a value in every branch of the `if...else` statement.

```
 int AddSquares(int n)
 {
   int i=0;
   int sum=0;

   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }

   /*Fix: Place a return statement on branches of if-else */
   else
     return 0;
  }
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

MSC52-CPP

Introduced in R2019a

CERT C++: PRE30-C

Do not create a universal character name through concatenation

Description

Rule Definition

Do not create a universal character name through concatenation.

Polyspace Implementation

This checker checks for **Universal character name from token concatenation**.

Examples

Universal character name from token concatenation

Issue

Universal character name from token concatenation occurs when two preprocessing tokens joined with a ## operator create a universal character name. A universal character name begins with `\u` or `\U` followed by hexadecimal digits. It represents a character not found in the basic character set.

For instance, you form the character `\u0401` by joining two tokens:

```
#define assign(uc1, uc2, val) uc1##uc2 = val
...
assign(\u04, 01, 4);
```

Risk

The C11 Standard (Sec. 5.1.1.2) states that if a universal character name is formed by token concatenation, the behavior is undefined.

Fix

Use the universal character name directly instead of producing it through token concatenation.

Example - Universal Character Name from Token Concatenation

```
#define assign(uc1, uc2, val) uc1##uc2 = val

int func(void) {
    int \u0401 = 0;
    assign(\u04, 01, 4);
    return \u0401;
}
```

In this example, the `assign` macro, when expanded, joins the two tokens `\u04` and `01` to form the universal character name `\u0401`.

Correction — Use Universal Character Name Directly

One possible correction is to use the universal character name `\u0401` directly. The correction redefines the `assign` macro so that it does not join tokens.

```
#define assign(ucn, val) ucn = val

int func(void) {
    int \u0401 = 0;
    assign(\u0401, 4);
    return \u0401;
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

PRE30-C

Introduced in R2019a

CERT C++: PRE31-C

Avoid side effects in arguments to unsafe macros

Description

Rule Definition

Avoid side effects in arguments to unsafe macros.

Polyspace Implementation

This checker checks for **Side effect in arguments to unsafe macro**.

Examples

Side effect in arguments to unsafe macro

Issue

Side effect in arguments to unsafe macro occurs when you call an unsafe macro with an expression that has a side effect.

- *Unsafe macro*: When expanded, an unsafe macro evaluates its arguments multiple times or does not evaluate its argument at all.

  For instance, the `ABS` macro evaluates its argument `x` twice.

  ```
  #define ABS(x) (((x) < 0) ? -(x) : (x))
  ```
- *Side effect*: When evaluated, an expression with a side effect modifies at least one of the variables in the expression.

  For instance, `++n` modifies `n`, but `n+1` does not modify `n`.

  The checker does not consider side effects in nested macros. The checker also does not consider function calls or volatile variable access as side effects.

Risk

If you call an unsafe macro with an expression that has a side effect, the expression is evaluated multiple times or not evaluated at all. The side effect can occur multiple times or not occur at all, causing unexpected behavior.

For instance, in the call `MACRO(++n)`, you expect only one increment of the variable `n`. If `MACRO` is an unsafe macro, the increment happens more than once or does not happen at all.

The checker flags expressions with side effects in the `assert` macro because the `assert` macro is disabled in non-debug mode. To compile in non-debug mode, you define the `NDEBUG` macro during compilation. For instance, in GCC, you use the flag `-DNDEBUG`.

Fix

Evaluate the expression with a side effect in a separate statement, and then use the result as a macro argument.

For instance, instead of:

```
MACRO(++n);
```

perform the operation in two steps:

```
++n;
MACRO(n);
```

Alternatively, use an inline function instead of a macro. Pass the expression with side effect as argument to the inline function.

The checker considers modifications of a local variable defined only in the block scope of a macro body as a side effect. This defect cannot happen since the variable is visible only in the macro body. If you see a defect of this kind, ignore the defect.

Example - Macro Argument with Side Effects

```
#define ABS(x) (((x) < 0) ? -(x) : (x))

void func(int n) {
  /* Validate that n is within the desired range */
  int m = ABS(++n);

  /* ... */
}
```

In this example, the ABS macro evaluates its argument twice. The second evaluation can result in an unintended increment.

Correction — Separate Evaluation of Expression from Macro Usage

One possible correction is to first perform the increment, and then pass the result to the macro.

```
#define ABS(x) (((x) < 0) ? -(x) : (x))

void func(int n) {
  /* Validate that n is within the desired range */
  ++n;
  int m = ABS(n);

  /* ... */
}
```

Correction — Evaluate Expression in Inline Function

Another possible correction is to evaluate the expression in an inline function.

```
static inline int iabs(int x) {
  return (((x) < 0) ? -(x) : (x));
}

void func(int n) {
  /* Validate that n is within the desired range */

int m = iabs(++n);

  /* ... */
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

PRE31-C

Introduced in R2019a

CERT C++: PRE32-C

Do not use preprocessor directives in invocations of function-like macros

Description

Rule Definition

Do not use preprocessor directives in invocations of function-like macros.

Polyspace Implementation

This checker checks for **Preprocessor directive in macro argument**.

Examples

Preprocessor directive in macro argument

Issue

Preprocessor directive in macro argument occurs when you use a preprocessor directive in the argument to a function-like macro or a function that might be implemented as a function-like macro.

For instance, a `#ifdef` statement occurs in the argument to a `memcpy` function. The `memcpy` function might be implemented as a macro.

```
memcpy(dest, src,
    #ifdef PLATFORM1
      12
    #else
      24
    #endif
  );
```

The checker flags similar usage in `printf` and `assert`, which can also be implemented as macros.

Risk

During preprocessing, a function-like macro call is replaced by the macro body and the parameters are replaced by the arguments to the macro call (argument substitution). Suppose a macro `min()` is defined as follows.

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

When you call `min(1,2)`, it is replaced by the body `((X) < (Y) ? (X) : (Y))`. `X` and `Y` are replaced by 1 and 2.

According to the C11 Standard (Sec. 6.10.3), if the list of arguments to a function-like macro itself has preprocessing directives, the argument substitution during preprocessing is undefined.

Fix

To ensure that the argument substitution happens in an unambiguous manner, use the preprocessor directives outside the function-like macro.

For instance, to execute `memcpy` with different arguments based on a `#ifdef` directive, call `memcpy` multiple times within the `#ifdef` directive branches.

```
#ifdef PLATFORM1
    memcpy(dest, src, 12);
#else
    memcpy(dest, src, 24);
#endif
```

Example - Directives in Function-Like Macros

```
#include <stdio.h>

#define print(A) printf(#A)

void func(void) {
    print(
#ifdef SW
          "Message 1"
#else
          "Message 2"
#endif
         );
}
```

In this example, the preprocessor directives `#ifdef` and `#endif` occur in the argument to the function-like macro `print()`.

Correction — Use Directives Outside Macro

One possible correction is to use the function-like macro multiple times in the branches of the `#ifdef` directive.

```
#include <stdio.h>

#define print(A) printf(#A)

void func(void) {
#ifdef SW
        print("Message 1");
#else
        print("Message 2");
#endif
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

PRE32-C

Introduced in R2019a

CERT C++: SIG31-C

Do not access shared objects in signal handlers

Description

Rule Definition

Do not access shared objects in signal handlers.

Polyspace Implementation

This checker checks for **Shared data access within signal handler**.

Examples

Shared data access within signal handler

Issue

Shared data access within signal handler occurs when you access or modify a shared object inside a signal handler.

Risk

When you define a signal handler function to access or modify a shared object, the handler accesses or modifies the shared object when it receives a signal. If another function is already accessing the shared object, that function causes a race condition and can leave the data in an inconsistent state.

Fix

To access or modify shared objects inside a signal handler, check that the objects are lock-free atomic, or, if they are integers, declare them as `volatile sig_atomic_t`.

Example - `int` Variable Access in Signal Handler

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* declare global variable. */
int e_flag;

void sig_handler(int signum)
{
    /* Signal handler accesses variable that is not
    of type volatile sig_atomic_t. */
    e_flag = signum;
}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
```

```
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

In this example, `sig_handler` accesses `e_flag`, a variable of type `int`. A concurrent access by another function can leave `e_flag` in an inconsistent state.

Correction — Declare Variable of Type `volatile sig_atomic_t`

Before you access a shared variable from a signal handler, declare the variable with type `volatile sig_atomic_t` instead of `int`. You can safely access variables of this type asynchronously.

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* Declare variable of type volatile sig_atomic_t. */
volatile sig_atomic_t e_flag;
void sig_handler(int signum)
{
    /* Use variable of proper type inside signal handler. */
    e_flag = signum;

}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics
“Check for Coding Standard Violations”

External Websites
SIG31-C

Introduced in R2019a

CERT C++: SIG34-C

Do not call signal() from within interruptible signal handlers

Description

Rule Definition

Do not call signal() from within interruptible signal handlers.

Polyspace Implementation

This checker checks for **Signal call from within signal handler**.

Examples

Signal call from within signal handler

Issue

Signal call from within signal handler occurs when you call `signal()` from a nonpersistent signal handler on a Windows platform.

Risk

A nonpersistent signal handler is reset after catching a signal. The handler does not catch subsequent signals unless the handler is reestablished by calling `signal()`. A nonpersistent signal handler on a Windows platform is reset to `SIG_DFL`. If another signal interrupts the execution of the handler, that signal can cause a race condition between `SIG_DFL` and the existing signal handler. A call to `signal()` can also result in an infinite loop inside the handler.

Fix

Do not call `signal()` from a signal handler on Windows platforms.

Example - `signal()` Called from Signal Handler

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;

void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;

    /* Call signal() to reestablish sig_handler
    upon receiving SIG_ERR. */

    if (signal(s0, sig_handler) == SIG_ERR)
```

```
    {
        /* Handle error */
    }
}

void func(void)
{
        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */

        }
  /* more code */
}
```

In this example, the definition of `sig_handler()` includes a call to `signal()` when the handler catches `SIG_ERR`. On Windows platforms, signal handlers are nonpersistent. This code can result in a race condition.

Correction — Do Not Call `signal()` from Signal Handler

If your code requires the use of a persistent signal handler on a Windows platform, use a persistent signal handler after performing a thorough risk analysis.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;


void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;
    /* No call to signal() */
}

int main(void)
{

        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */

        }
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

SIG34-C

Introduced in R2019a

CERT C++: SIG35-C

Do not return from a computational exception signal handler

Description

Rule Definition

Do not return from a computational exception signal handler.

Polyspace Implementation

This checker checks for **Return from computational exception signal handler**.

Examples

Return from computational exception signal handler

Issue

Return from computational exception signal handler occurs when a signal handler returns after catching a computational exception signal `SIGFPE`, `SIGILL`, or `SIGSEGV`.

Risk

A signal handler that returns normally from a computational exception is undefined behavior. Even if the handler attempts to fix the error that triggered the signal, the program can behave unexpectedly.

Fix

Check the validity of the values of your variables before the computation to avoid using a signal handler to catch exceptions. If you cannot avoid a handler to catch computation exception signals, call `abort()`, `quick_exit()`, or `_Exit()` in the handler to stop the program.

Example - Signal Handler Return from Division by Zero

```
#include <errno.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>

static volatile sig_atomic_t denom;
/* Declare signal handler to catch division by zero
computation error. */
void sig_handler(int s)
{
    int s0 = s;
    if (denom == 0)
    {
        denom = 1;
    }
    /* Normal return from computation exception
    signal */
    return;
}
```

```
long func(int v)
{
    denom = (sig_atomic_t)v;

        if (signal(SIGFPE, sig_handler) == SIG_ERR)
        {
            /* Handle error */
        }

    long result = 100 / (long)denom;
    return result;
}
```

In this example, `sig_handler` is declared to handle a division by zero computation error. The handler changes the value of `denom` if it is zero and returns, which is undefined behavior.

Correction — Call `abort()` to Terminate Program

After catching a computational exception, call `abort()` from `sig_handler` to exit the program without further error.

```
#include <errno.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>

static volatile sig_atomic_t denom;
/* Declare signal handler to catch division by zero
computation error. */

void sig_handler(int s)
{
    int s0 = s;
    /* call to abort() to exit the program */
    abort();
}

long func(int v)
{
    denom = (sig_atomic_t)v;

        if (signal(SIGFPE, sig_handler) == SIG_ERR)
        {
            /* Handle error */
        }

    long result = 100 / (long)denom;
    return result;
}
```

Check Information

Group: 49. Miscellaneous (MSC)

See Also

`Check SEI CERT-C++ (-cert-cpp))`

Topics

“Check for Coding Standard Violations”

External Websites

SIG35-C

Introduced in R2019a

AUTOSAR C++14 Rules

AUTOSAR C++14 Rule A0-1-2

The value returned by a function having a non-void return type that is not an overloaded operator shall be used.

Description

Rule Definition

The value returned by a function having a non-void return type that is not an overloaded operator shall be used.

Rationale

The unused return value might indicate a coding error or oversight.

Overloaded operators are excluded from this rule because their usage must emulate built-in operators which might not use their return value.

Polyspace Implementation

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Return Value Not Used

```
#include <iostream>
#include <new>

int assignMemory(int * ptr){
     int res = 1;
     ptr = new (std::nothrow) int;
     if(ptr==NULL) {
         res = 0;
     }
     return res;
}

void main() {
    int val;
    int status;

    assignMemory(&val);   //Noncompliant
    status = assignMemory(&val); //Compliant
    (void)assignMemory(&val); //Compliant

}
```

The first call to the function `assignMemory` is noncompliant because the return value is not used. The second and third calls use the return value. The return value from the second call is assigned to a local variable.

The return value from the third call is cast to `void`. Casting to `void` indicates deliberate non-use of the return value and cannot be a coding oversight.

Check Information

Group: Language Independent Issues

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A0-1-4

There shall be no unused named parameters in non-virtual functions.

Description

Rule Definition

There shall be no unused named parameters in non-virtual functions.

Rationale

Unused parameters can indicate that the code is possibly incomplete. The parameter is possibly intended for an operation that you forgot to code or leftover from a design change.

If the parameters are obtained by copy and the copied objects are large, the redundant copies can slow down performance.

Polyspace Implementation

The checker flags a function that has unused named parameters unless the function body is empty.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language Independent Issues

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A0-1-5

There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it.

Description

Rule Definition

There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it.

Rationale

Unused parameters can indicate that the code is possibly incomplete. The parameter is possibly intended for an operation that you forgot to code.

The rule focuses on virtual functions because all functions that override a virtual function must have the same signature as the virtual function, including number and type of parameters. If a parameter is indeed not required, the issue can cascade from the original function to all overriding functions.

However, in an overriding function, you might not have need for a certain parameter. You can leave that parameter unnamed. This rule enforces the convention that unused parameters stay unnamed.

Polyspace Implementation

For each virtual function, the checker looks at all overrides of the function. If an override has a named parameter that is not used, the checker shows a violation on the original virtual function and lists the override as a supporting event.

Note that Polyspace checks for unused parameters in virtual functions within single translation units. For instance, if a base class contains a virtual method with an unused parameter but the derived class implementation of the method uses that parameter, the rule is not violated. However, if the base class and derived class are defined in different files, the checker, which operates file by file, flags a violation of this rule on the base class.

The checker does not flag unused parameters in functions with empty bodies.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language independent issues

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A0-1-6

There should be no unused type declarations.

Description

Rule Definition

There should be no unused type declarations.

Rationale

If a type is declared but not used, when reviewing the code later, it is unclear if the type is redundant or left unused by mistake.

Unused types can indicate coding errors. For instance, you declared a enumerated data type for some specialized data but used an integer type for the data.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Unused `enum` Declaration

```
enum switchValue {low, medium, high}; //Noncompliant

void operate(int userInput) {
    switch(userInput) {
        case 0: // Turn on low setting
                 break;
        case 1: // Turn on medium setting
                 break;
        case 2: // Turn on high setting
                 break;
        default: // Return error
    }
}
```

In this example, the enumerated type `switchValue` is not used. Perhaps the intention was to use the type as `switch` input like this.

```
enum switchValue {low, medium, high}; //Compliant

void operate(switchValue userInput) {
    switch(userInput) {
        case low: // Turn on low setting
                   break;
        case medium: // Turn on medium setting
                     break;
        case high: // Turn on high setting
                   break;
        default: // Return error
```

```
    }
}
```

Check Information

Group: Language Independent Issues

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A0-4-2

Type long double shall not be used

Description

Rule Definition

Type long double shall not be used.

Rationale

The size of `long double` is implementation-dependent and reduces the portability of your code across compilers. Compilers can implement `long double` as a synonym for `double` or an 80-bit extended precision type or 128-bit quadruple precision type that are more precise than `double`.

Instead, for multiple precision arithmetic that requires types more precise than `double`, use libraries that support multiple precision arithmetic with well-defined data types.

Polyspace Implementation

The rule checker flags all uses of the `long double` keyword.

If you do not want to fix the issue, add a comment justifying the result. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `long double` Keyword

```
void func() {
  float f{0.1F};  //Compliant
  double D(0.1);  //Compliant
  long double LD(0.1L);  //Noncompliant
}
```

The use of `long double` violates this rule.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A10-1-1

Class shall not be derived from more than one base class which is not an interface class.

Description

Rule Definition

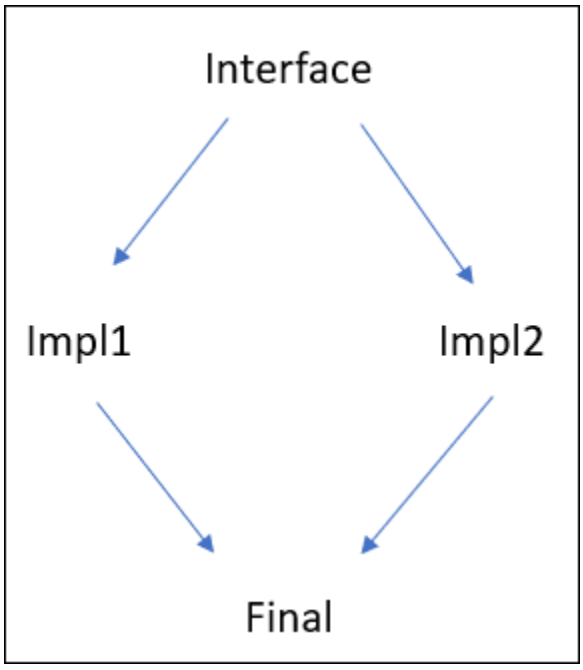
Class shall not be derived from more than one base class which is not an interface class.

Rationale

If a class inherits from multiple non-interface classes, the class essentially has access to multiple implementations. Maintaining the code can be difficult.

When a class inherits from multiple non-interface classes, there is a likelihood that the same member function exists in those base classes and must be overridden in the derived class. The likelihood increases when those base classes themselves inherit from a common base class (diamond structure).

Suppose, an interface class `Interface` has two concrete implementations, `Impl1` and `Impl2`, and a class `Final` derives from both implementations. The class hierarchy has this diamond structure.



The following issues can occur:

- *Overrides required in final derived class for disambiguation*:

  Both implementations `Impl1` and `Impl2` have a copy of all methods of the class `Interface`. To disambiguate which copy can be called through a `Final` object, you typically create yet another override of all methods in the `Final` class where you call both copies explicitly using the scope resolution operator `::` (or one copy, if you choose). See example below.

Each time you add a new pure virtual function to the class `Interface`, you have to not only create implementations in the immediate derived classes but also keep track of the entire class hierarchy and create overrides of those implementations in the class `Final`.

If the original class `Interface` is not an interface class, the problem is even more acute. Unless the inheritances are virtual, two copies of the methods of `Interface` are *implicitly* made in `Impl1` and `Impl2` (the diamond problem).

- *Final derived class responsible for initializing all classes in hierarchy*:

  To avoid double initializations in multiple inheritance, the C++ standard requires that you call the constructors of all previous classes in the most derived class.

  In the preceding example, the `Final` class constructor not only has to call the constructors of `Impl1` and `Impl2` but also the constructor of their parent class `Interface`. You have to trace beyond the immediate parents to determine which constructors to call in the final derived class.

These problems disappear if multiple inheritances are restricted to situations where a class can derive from multiple classes but only one of them can be a non-interface class. An interface class is a class that has only pure virtual functions and data members that are compile-time constants (static, `contexpr`-s). The class has no state and its sole purpose is to be implemented by derived classes.

Multiple inheritance was designed for situations where a class extends one concrete implementation but also implements other ideas represented by interface classes. Other uses of multiple inheritance can lead to maintenance hazards.

Polyspace Implementation

The checker flags multiple inheritances where more than one base class is a non-interface class.

An interface class is one that has only pure virtual functions and data members that are compile-time constants (static, `contexpr`-s).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Multiple Inheritance from Non-interface Classes

```
class Interface {
    public:
    virtual void setVal()=0;
};

class Impl1: public Interface{
    int val1;
public:
    void setVal() {
        val1 = 0;
    }
};

class Impl2: public Interface{
    int val2;
```

```
public:
    void setVal() {
        val2 = 0;
    }
};

class Final: public Impl1, public Impl2 { //Noncompliant
public:
    void setVal() {
        Impl1::setVal();
        Impl2::setVal();
    }

};

void main() {
    Final finalObj;
    finalObj.setVal();
}
```

In this example, the class final derives from classes `Impl1` and `Impl2`. Both classes `Impl1` and `Impl2` have data members that are not compile-time constants and member functions that are not pure virtual functions. Therefore, the classes are non-interface classes. Inheriting from two non-interface classes causes a coding rule violation.

Check Information

Group: Derived classes

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A10-2-1

Non-virtual member functions shall not be redefined in derived classes.

Description

Rule Definition

Non-virtual member functions shall not be redefined in derived classes.

Polyspace Implementation

Does not report for destructor.

Message in report file:

Inherited nonvirtual function `%s` shall not be redefined in a derived class.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A10-3-1

Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final.

Description

Rule Definition

Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final.

Rationale

Virtual functions implement polymorphic behavior in a class hierarchy. Once you declare a function as `virtual` in a base class, all instances of the function with an identical parameter list in the derived classes override the base function implicitly. If you rely on this implicit action by the compiler for implementing polymorphic functions, it can lead to errors. For instance:

- A function can become inadvertently `virtual` because its signature matches a virtual function in the base class.
- A function can become inadvertently non-virtual because there are differences in the parameter list.

Implicitly declaring virtual functions can also make the code hard to read.

To avoid inadvertent errors and to enhance readability, use the specifiers `virtual`, `override`, or `final` to explicitly define virtual or overriding functions. Because using more than one of these specifiers in a declaration is either redundant or a source of error, use exactly one of these specifiers:

- Only `virtual` to declare a new virtual function.
- Only `override` to declare a non-final overriding function of a virtual function.
- Only `final` to declare a final overriding function of a virtual function.

Polyspace Implementation

Polyspace flags declaration of virtual functions if:

- The declaration uses none of the specifiers.
- The declaration uses more than one of the specifiers.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use Exactly One Specifier to Declare Virtual Functions

```
#include<cstdint>
class Base
```

```
{
public:
    virtual void F() noexcept = 0;                  // Compliant
    virtual void G() noexcept final = 0;            // Noncompliant
    virtual void H() noexcept final {}              // Noncompliant
    virtual void J() noexcept {}                    // Compliant
    virtual void K() noexcept {}                    // Compliant
    virtual ~Base() {}                              // Compliant
    virtual void M() noexcept {}                    // Compliant
    virtual void Z() noexcept {}                    // Compliant
    virtual void X() throw() {}                     // Compliant
    virtual void Y() noexcept {}                    // Compliant
};

class Derived : public Base
{
public:
    ~Derived() {}                                   // Noncompliant
    virtual void F() noexcept override {}           // Noncompliant
    void K() noexcept override final {}             // Noncompliant
    virtual void M() noexcept {}                    // Compliant
    void Z() noexcept override {}                   // Compliant
    void J() noexcept {}                            // Noncompliant
    void J(int) noexcept {}                         // Compliant
    virtual void X() throw() final {}               // Noncompliant
    virtual void Y() noexcept override final {}     // Noncompliant
};
class DD: public Derived{
//    void J(int) noexcept override{}       //Compilation error
};
main(){
    //...
}
```

- The destructor of the derived class `~Derived()` is a virtual function. Its declaration violates this rule because the declaration contains none of the three specifiers for virtual functions.
- The declaration of the pure virtual function `Base::G()` also violates this rule because the declaration contains both `virtual` and `final` as specifiers. A pure virtual function that is also specified as `final` is redundant.
- The declaration of the virtual function `Derived::J()` violates this rule because `Derived::J()` implicitly overrides the virtual function `Base::J()` without using the specifier `override`.
- The declarations of the virtual functions `Derived::X()` and `Derived::Y()` violate this rule because their declarations use more than one specifier.

The declaration of the function `DD::J(int)` produces a compilation error because `DD::J(int)` is trying to override `Derived::J(int)`. Because `Derived::J(int)` has a different signature than `Base::J()`, perhaps by error, `Derived::J(int)` is no longer a virtual function. Attempting to override `Derived::J(int)` by `DD::J(int)` results in a compilation error. Using exactly one specifier in the declaration of virtual functions can help detect errors.

Check Information

Group: Derived classes

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A10-3-2

Each overriding virtual function shall be declared with the override or final specifier.

Description

Rule Definition

Each overriding virtual function shall be declared with the override or final specifier.

Rationale

Virtual functions implement polymorphic behavior in a class hierarchy. Once you declare a function as `virtual` in a base class, all instances of the function with an identical parameter list in the derived classes override the base function implicitly. If you rely on this implicit action by the compiler for implementing polymorphic functions, it can lead to errors. For instance:

- A function can become inadvertently `virtual` because its signature matches a virtual function in the base class.
- A function can become inadvertently non-virtual because there are differences in the parameter list.

Implicitly declaring overriding virtual functions can also make the code hard to read.

To avoid inadvertent errors and to enhance readability, use the specifiers `override` and `final` explicitly in every declaration and definition of overriding functions.

Polyspace Implementation

Polyspace flags declarations and definitions of `virtual` functions if all of these statements are true:

- The function is in a derived class.
- The signature of the function matches the signature of a virtual function in the base class.
- The declaration of the function lacks the specifier `override` or `final`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Declare Overriding Virtual Functions by Using `override` or `final` Specifier

```
#include <cstdint>
class Base
{
public:
    virtual ~Base() {}
    virtual void F() noexcept = 0;
    virtual void G() noexcept {}
    virtual void Z() noexcept {}
    virtual Base& operator+=(Base const& oth) = 0;
```

```
};
class Derived1 : public Base
{
public:
    ~Derived1() override {}                          //Compliant
    void F() noexcept{}                              //Noncompliant
    virtual void G() noexcept {}                     //Noncompliant
    void Z() noexcept override {}                    // Compliant
    Derived1& operator+=(Base const& oth) override  // Compliant
    {
        return *this;
    }
};
class Derived2 : public Base
{
public:
    ~Derived2() {}                                   //  Noncompliant
    void F() noexcept override {}                    //  Compliant
    void G() noexcept override {}                    //  Compliant
    void Z() noexcept override {}                    //  Compliant
    Derived2& operator+=(Base const& oth)            //  Noncompliant
    {
        return *this;
    }
};
class Derived3 : public Base
{
    void F() noexcept override;                      // Compliant
};
void Derived3::F() noexcept{                            //Noncompliant
    //...
}

main(){

}
```

- The declaration of the function `Derived::F()` is flagged because its signature matches the signature of `Base::F()` and its declaration does not contain `override` or `final`.
- The declaration of the function `Derived::G()` is flagged because its signature matches the signature of `Base::G()` and its declaration does not contain `override` or `final`, even though the declaration uses the specifier `virtual`.
- The declaration of the function `Derived3::F()` in class `Derived3` uses the specifier `override`. Its definition does not use any specifier. Polyspace flags the definition of the function `Derived3::F()`.

Check Information

Group: Derived classes

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A10-3-3

Virtual functions shall not be introduced in a final class.

Description

Rule Definition

Virtual functions shall not be introduced in a final class.

Rationale

Declaring a function as `virtual` indicates that you intend to override the function in a derived class with a different implementation. The same function can then interact differently with different classes of a hierarchy. When you explicitly specify a class as `final`, you cannot derive a class from it. Because you cannot derive classes from a `final` class, do not introduce virtual functions in a `final` class. Specify all virtual functions in a `final` class by using the specifier `final`.

Polyspace Implementation

Polyspace flags the declaration of virtual functions in a `final` class that are declared with these specifiers:

- `virtual`
- `override`

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Specify Virtual Function in `final` Classes by Using `final`

```
#include <cstdint>
class Base
{
public:
    virtual ~Base() = default;
    virtual void F() noexcept = 0;
    virtual void G() noexcept {/*...*/}
};
class Derived final : public Base
{
public:

    void G() noexcept override{/*...*/} //Noncompliant
    virtual void Z() noexcept{/*...*/}  //Noncompliant
    virtual void H() noexcept = 0;    //Noncompliant
    void F() noexcept final{/*...*/}    //Compliant
};

main(){
```

```
}
```

The functions `Derived::G()`, `Derived::Z()`, and `Derived::H()` are `virtual` functions that are not specified as `final`. Their declarations indicate that some functions in a derived class might override these functions. The class `Derived` is specified as `final`. That is, there are no derived classes from this class. The declarations of `Derived::G()`, `Derived::Z()`, and `Derived::H()` are inconsistent with the declaration of their class `Derived`. Polyspace flags the declarations of the functions. The function `Derived::F()` is declared as `final`. This declaration complies with this rule.

Check Information

Group: Derived classes

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A10-3-5

A user-defined assignment operator shall not be virtual.

Description

Rule Definition

A user-defined assignment operator shall not be virtual.

Rationale

Defining an assignment operator as `virtual` in a base class indicates that you want to override it in the derived classes. Overriding the assignment operator in derived classes can lead to undefined behavior and run-time errors. Consider this code snippet where a virtual assignment operator is overridden in two derived classes.

```
class Base {public:
    virtual Base& operator=(Base const& oth) = 0;
    //...
};
class Derived public: Base{ public:
    Derived& operator=(Base const& oth) override{/*...*/}
    //...
};
class Derived2 public: Base{public:
    Derived2& operator=(Base const& oth) override{/*...*/}
    //...
};
main(){
    Derived d1;
       Derived2 d2;
    d1 = d2;
}
```

Because `Derived::operator=` and `Derived2::operator=` overrides `Base::operator=`, their parameter lists must be identical.

- `Derived::operator=` takes reference to a `Base` object as input and returns a reference to `Derived`.
- `Derived2::operator=` takes reference to a `Base` object as input and returns a reference to `Derived2`.

The `Derived::operator=` accepts references to both `Base` and `Derived` class objects because references to derived classes are type-compatible with their base classes. Similarly, the `Derived2::operator=` also accepts references to both `Base` and `Derived2` class objects. Assigning a `Derived` object to a `Derived2` object in `d1=d2` produces no compilation error. The objects `d1` and `d2` are unrelated. Assigning, copying, or moving operations between such unrelated objects are undefined and can lead to run-time errors.

To avoid undefined behavior and run-time errors, keep user-defined assignment operators as non-virtual. This rule applies to these operators:

- Assignment
- Copy and move assignment
- All compound assignment

Polyspace Implementation

Polyspace flags the declaration of any virtual assignment operators in a base class.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Avoid Defining Assignment Operators as `virtual`

This example shows how Polyspace flags `virtual` assignment operators.

```
#include <cstdint>
class Base
{
  public:
    virtual Base& operator=(Base const& oth) = 0;   // Noncompliant
    virtual Base& operator+=(Base const& rhs) = 0; // Noncompliant
};
class Derived : public Base
{
  public:
    Derived& operator=(Base const& oth) override
    {
      return *this;
    }
    Derived& operator+=(Base const& oth) override
    {
      return *this;
    }
    Derived& operator-=(Derived const& oth) // Compliant
    {
      return *this;
    }
};
class Derived2 : public Base
{
  public:

    Derived2& operator=(Base const& oth) override
    {
      return *this;
    }
    Derived2& operator+=(Base const& oth) override
    {
      return *this;
    }
    Derived2& operator-=(Derived2 const& oth)   // Compliant
    {
      return *this;
```

```
    }
};
/*
*/
void Fn() noexcept
{
  Derived b;
  Derived2 c;
  b = c;
  b += c;
  c = b;
  c += b;
  // b -= c; // Compilation error
  // c -= b; // Compilation error

}
```

The classes `Derived` and `Derived2` are derived from `Base`. In the `Base` class, the assignment operators `Base::operator=` and `Base::operator+=` are declared as virtual. None of the following cause compilation errors:

- You can assign the `Derived` object `b` to `Derived2` object `c` and vice versa.
- You can add the `Derived` object `b` to `Derived2` object `c`. You can assign the result to either `b` or `c`.

Because `b` and `c` are unrelated objects, all of the preceding behaviors are undefined and can cause run-time errors. Declaring the `Base::operator=` and `Base::operator+=` as `virtual` eventually lead to the undefined behaviors. Polyspace flags these virtual assignment operators.

The declaration of `Base::operator-=` is non-virtual. Operations such as `b-=c` and `c-=b` cause compilation errors.

Check Information

Group: Derived classes

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A1-1-1

All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features.

Description

Rule Definition

All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features.

Polyspace Implementation

The checker reports compilation errors as detected by a compiler that strictly adheres to the C++03 Standard (ISO/IEC 14882:2003).

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: General

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A11-0-2

A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class.

Description

Rule Definition

A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class.

Rationale

The items prohibited by this rule are not supported for `struct` types in C code. In C++, a `struct` type can have private data members, member functions, be inherited and inherit from other `struct`-s or `class`-es. However, a developer typically associates these features with a `class` type. Adhering to this rule makes sure that you use only classes to implement object oriented concepts such as data encapsulation and inheritance.

Adhering to this rule also makes sure that your `struct` types conform to the rules of Plain Old Data (POD) types and can be exchanged with C code.

Polyspace Implementation

The checker flags `struct` types with one or more of these features:

- Contains private or protected data members.

  `struct` members are public by default.
- Contains member functions.
- Acts as base class for another `struct` or `class`, or inherits from another `struct` or `class`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Struct Types with Class-Like Features

```
#include <cstdint>
#include <iostream>

struct loginCredentials1 { //Noncompliant: Private members
    int32_t username;
private:
    int32_t pwd;
};
```

```
struct loginCredentials2 {   //Noncompliant: Member functions
    int32_t username;
    int32_t pwd;
    void readFromFile(std::string fileName) {
        //Read members data from file
    }
};

struct loginCredentials3 {  //Noncompliant: Acts as base for another struct
    int32_t username;
    int32_t pwd;
};

struct adminLoginCredentials: loginCredentials3 { //Noncompliant: Inherits from another struct
    std::string permissions;
};
```

In this example, all `struct` types are noncompliant.

- `loginCredentials1` contains a private data member `pwd`.
- `loginCredentials2` contains a member function `readFromFile()`.
- `loginCredentials3` acts as a base for the struct `adminLoginCredentials`.
- `adminLoginCredentials` inherits from the struct `loginCredentials3`.

Check Information

Group: Member access control

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A11-3-1

Friend declarations shall not be used

Description

Rule Definition

Friend declarations shall not be used.

Rationale

You declare a function as friend of a class to access private members of the class outside the class scope.

```
class A
{
  int data;
  public:
    // operator+ can access private members of class A such as data
    friend A const operator+(A const& lhs, A const& rhs);
};
```

Friend functions and friend classes reduce data encapsulation. Private members of a class are no longer accessible only through the class methods.

Code with friend functions can be difficult to maintain. For instance, if class `myClass` has a friend class `anotherClass`, when you change a data member of `myClass`, you have to find all instances of its usage in member functions of `anotherClass`.

Polyspace Implementation

The rule checker flags all uses of the `friend` keyword.

The checker follows specifications of AUTOSAR C++ 14 release 18-03 (March 2018). However, release 18-10 and later releases of AUTOSAR C++14 allows an exception for comparison operators such as `operator==`. If the rule checker flags the use of comparison operators, add a comment justifying the result. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `friend` Keyword

```
class myClass
{
  int data;
  public:
    myClass& operator+=(myClass const& oth);
    friend myClass const operator+(myClass const& lhs, myClass const& rhs);
```

```
    // Noncompliant: Use of friend keyword
};
```

`operator+` is a friend function of class `myClass` and can access its private member, `data`. The presence of this friend function violates the rule.

Check Information

Group: Member Access Control

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A12-0-1

If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well.

Description

Rule Definition

If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well.

Rationale

These special member functions are called for copy or move operations:

- Copy constructor
- Copy assignment operator
- Move constructor
- Move assignment operator
- Destructor

If you do not explicitly declare any of these functions, the compiler defines them implicitly. This implicit definition implements shallow copying of objects and can cause errors. If you need to explicitly declare any of the special member functions, you must declare all of them. For instance, suppose you want to copy an object that contains a raw pointer to a dynamically allocated memory. The implicit copy constructor shallow-copies the object, after which the original pointer and the copied pointer point to the same memory. If one of the objects is destroyed, the allocated memory is deallocated, leaving a dangling pointer in the other object. Accessing the dangling pointer can cause segmentation errors. Because all the special member functions are closely related, the implicit implementation of the other functions can lead to similar errors. To manage the life cycle of the dynamically allocated resource, explicitly declare all five of the special member functions (Rule of Five). Alternatively, you can use objects where memory management is correctly implemented in the implicit definition of the special member functions and explicitly declare none of them (Rule of Zero).

When you explicitly declare some but not all of the special member functions, the compiler can prevent the use of the undeclared special member functions. For example, if you explicitly declare only the copy constructor or destructor functions of a class, the compiler no longer defines the move constructor and move assignment operator implicitly. The class becomes a copy-only class, perhaps inadvertently. Conversely, if you explicitly declare only the move constructor and move assignment operator, the compiler disables the copy constructor and copy assignment operator by defining them as deleted. The class becomes a move-only class, which might not have been your intention. To avoid such unwanted effects, either follow the Rule of Five or follow the Rule of Zero.

The constructor of a class is not part of this rule.

Polyspace Implementation

Polyspace flags classes that explicitly declare some but not all of the five special member functions. Note that the move constructor and move assignment operators were introduced in C++11. Polyspace does not make any exception for older codes.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Follow Either The Rule of Five or The Rule of Zero

This example demonstrates the Polyspace implementation of AUTOSAR rule A12-0-1.

```
// Class rendered copy-only, perhaps inadvertently
class A               // Noncompliant.
{
    public:
      ~A()
      {
        // ...
      }

    private:
      // Member data ...
};


//Class rendered move-only, perhaps inadvertently
class B          // Noncompliant
{
   public:
        B(B&&) = default;
        B& operator=(B&&) = default;
    private:
      // Member data ...

};
template<typename T>
class BaseT              // Compliant - rule of five.
{
  public:
    BaseT(BaseT const&) = delete;
    BaseT(BaseT&&) = delete;
    virtual ~BaseT() = default;
    BaseT& operator=(BaseT const&) = delete;
    BaseT& operator=(BaseT&&) = delete;
    protected:
    BaseT() = default;
};

template<typename T>
class SimpleT           // Compliant - rule of zero.
{
  public:
```

```
    SimpleT(T t): t_(t)
    {

    }

  private:
    T t_;
};

main()
{
    //..
}
```

The class `A` declares only its destructor, which makes this class copy-only because the compiler no longer defines the move constructor and move assignment operator. The class `B` declares the move constructor and the move assignment operator, which makes this class move-only because the compiler disables the copy constructors and copy assignment operators. It is not clear whether these effects are deliberate. Polyspace flags these declarations and indicates which special member functions are missing. The class `BaseT` is compliant with this rule because all five of the special member functions are declared. Similarly, `SimpleT` is compliant because it declares none of the special member functions and relies on their implicit definition.

Check Information

Group: Special member functions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A12-1-1

Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members.

Description

Rule Definition

Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Special Member Functions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A12-1-4

All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Description

Rule Definition

All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Special Member Functions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A12-4-1

Destructor of a base class shall be public virtual, public override or protected non-virtual.

Description

Rule Definition

Destructor of a base class shall be public virtual, public override or protected non-virtual.

Rationale

If a base class destructor is not public virtual or public override, the class cannot behave polymorphically for deletion of derived class objects.

If a pointer to a base class refers to a derived class object and you use the pointer to delete the object:

```
class Base {
  public:
    ~Base() {}
};

class Derived: public Base {
  public:
    ~Derived() {}
};
...
void func(Base* ptr) {
    //ptr might point to a Base or Derived object
    delete ptr;
}
```

only the base class destructor is called. Additional resources allocated in the derived class are not released and can cause a resource leak. See example below.

If you want to prevent calling the derived class destructor through a base class pointer, make your intent explicit by making the destructor protected. Otherwise, it might appear that the possibility of polymorphic deletion of derived class objects was not considered.

Polyspace Implementation

The checker flags base classes with destructors that are not public virtual, public override or protected non-virtual.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Base Class Destructor Not Virtual

```
#include <new>
```

```
class Base {
  public:
    Base() {}
    ~Base() {} //Noncompliant
};

class Derived: public Base {
     int *arr;
  public:
     Derived() {
        arr = new int(5);
     }
    ~Derived() {
        delete arr;
     }
};

void main() {
    Base* basePtr = new Derived();
    delete basePtr;
}
```

In this example, the class `Base` has a non-virtual destructor. As a result, when the pointer `basePtr` is deleted, only the destructor of class `Base` is invoked. However, `basePtr` points to an object of class `Derived`. The deletion is not complete because the destructor of class `Derived` is not invoked. In particular, the data member `arr` in the derived object is not deleted.

Check Information

Group: Special member functions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A12-6-1

All class data members that are initialized by the constructor shall be initialized using member initializers.

Description

Rule Definition

All class data members that are initialized by the constructor shall be initialized using member initializers.

Polyspace Implementation

All data should be initialized in the initialization list except for array. Does not report that an assignment exists in `ctor` body.

Message in report file:

Initialization of nonstatic class members "*<field>*" will be performed through the member initialization list.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A12-8-5

A copy assignment and a move assignment operators shall handle self-assignment.

Description

Rule Definition

A copy assignment and a move assignment operators shall handle self-assignment.

Polyspace Implementation

Reports when copy assignment body does not begin with "`if (this != arg)`"

A violation is not raised if an empty `else` statement follows the `if`, or the body contains only a return statement.

A violation is raised when the `if` statement is followed by a statement other than the return statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
"Check for Coding Standard Violations"

Introduced in R2019a

AUTOSAR C++14 Rule A12-8-6

Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class.

Description

Rule Definition

Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class.

Rationale

Pointers to derived classes are type-compatible with pointers to base classes. A pointer can be an object of the base class while pointing to an object of the derived class. When such an object is copied, the base copy constructor is invoked and the copied object has only the base part of the original object. To avoid inadvertent slicing during copy and move, suppress these operations in the base class by:

- Declaring copy and move constructors and copy assignment and move assignment operators as `protected`.
- Defining copy and move constructors and copy assignment and move assignment operators as "`=delete`".

Polyspace Implementation

Polyspace flags these special member functions of a base class when they are not declared `protected` or defined as `=delete`:

- Copy constructor
- Move constructor
- Copy assignment operator
- Move assignment operator

Polyspace indicates which special member function violates this rule.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Suppress Copy and Move Operations in Base Classes

```
#include <cstdint>
#include <memory>
#include <utility>
#include <vector>
class A
{
```

```
  public:
    int base_var;
    A() = default;
    A(A const&) = default;                //Noncompliant
    A(A&&) = default;                     //Noncompliant
    virtual ~A() = 0;
    A& operator=(A const&) = default;     //Noncompliant
    A& operator=(A&&) = default;          //Noncompliant
};
class B : public A
{
    int derived_var;
};
class C //
{
  public:
    int base_var;
    C() = default;
    virtual ~C() = 0;

  protected:
    C(C const&) = default;                //Compliant
    C(C&&) = default;                     //Compliant
    C& operator=(C const&) = default;     //Compliant
    C& operator=(C&&) = default;          //Compliant
};
class D : public C
{
    int derived_var;
};
class E
{
  public:
    int base_var;
    E() = default;
    virtual ~E() = default;
    E(E const&) = delete;                   //Compliant
    E(E&&) = delete;                        //Compliant
    E& operator=(E const&) = delete;        //Compliant
    E& operator=(E&&) = delete;             //Compliant
};

class F : public E
{
    int derived_var;
};
void Fn1() noexcept
{
  B obj1;
  B obj2;
  A* ptr1 = &obj1;
  A* ptr2 = &obj2;
  *ptr1 = *ptr2;             // Partial assignment only
  *ptr1 = std::move(*ptr2); // Partial move only
  D obj3;
  D obj4;
  C* ptr3 = &obj3;
  C* ptr4 = &obj4;
```

```
  //*ptr3 = *ptr4; // Compilation error
  //*ptr3 = std::move(*ptr4); // Compilation error
  F obj5;
  F obj6;
  E* ptr5 = &obj5;
  E* ptr6 = &obj6;
  //*ptr5 = *ptr6; // Compilation error
  //*ptr5 = std::move(*ptr6); // Compilation error
}
```

The Class `A` is a base class with default copy and move constructors and default copy and move assignment operators. The class `B` is derived from `A` and has a variable `derived_var` that is absent in `A`. In `Fn1()`, two pointers `ptr1` and `ptr2` are created. They are objects of the base class `A`, but point to `obj1` and `obj2` respectively, which are objects of the derived class `B`. The assignment `A *ptr = &obj1;` is an example of polymorphic behavior where you can declare a pointer of the base class and assign objects of any derived class to it.

Because `ptr1` and `ptr2` are objects of the base class A, the copy operation in `*ptr1 = *ptr2` invokes the `default` copy assignment operator of class `A`. The `default` semantics copies only the base part of `obj2` into `obj1`. That is, `obj2.derived_var` is not copied into `obj1.derived_var`. Similarly, the ownership of `obj2.derived_var` is not moved to `obj1` by the move operation in `*ptr1 = std::move(*ptr2)`. To avoid inadvertent slicing, suppress the copy and move operations in the base class of a class hierarchy. Polyspace flags the copy and move functions in base class `A` because these functions are neither declared as `protected` nor defined as `=delete`.

In class `C`, the copy and move functions are suppressed by declaring the copy and move constructors and copy assignment and move assignment operators `protected`. In class `E`, the copy and move operations are suppressed by declaring thesespecial member functions as `=delete`. If you invoke the copy or move operations of these base classes, the compiler generated an error. The definitions of the base classes `C` and `E` are compliant with this rule.

Check Information

Group: Special member functions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A13-1-2

User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters.

Description

Rule Definition

User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters.

Rationale

Since C++11, you can add suffixes to literals that convert numeric values under the hood. For instance, in code where you perform all calculations in a common unit, you can leave unit conversions to dedicated operators and simply use literal suffixes for the units when defining constant values.

In this example, the literal suffixes `_m` and `_km` resolve to calls to `operator"" _m()` and `operator"" _km()` respectively. The operators ensure that all values are converted to the same unit.

```
constexpr long double operator"" _m(long double metres) {
    return metres;
}

constexpr long double operator"" _km(long double kilometres) {
    return 1000*kilometres;
}
...
long double minSteps = 100.0_m;
long double interCityDist = 100.0_km;
```

User defined literal suffixes must begin with an underscore (_). Literal suffixes not beginning with underscore are reserved for members of the standard library.

Polyspace Implementation

The rule checker flags definitions of the form:

```
operator "" suffix
```

where *`suffix`* does not begin with an underscore or following the underscore, contains characters other than letters (numbers, special characters, and so on).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Overloading

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A13-2-1

An assignment operator shall return a reference to "this".

Description

Rule Definition

An assignment operator shall return a reference to "*this*"*.*

Polyspace Implementation

The following operators should return `*this` on method, and `*first_arg` on plain function:

- `operator=`
- `operator+=`
- `operator-=`
- `operator*=`
- `operator >>=`
- `operator <<=`
- `operator /=`
- `operator %=`
- `operator |=`
- `operator &=`
- `operator ^=`
- Prefix `operator++`
- Prefix `operator--`

Does not report when no return exists.

No special message if type does not match.

Messages in report file:

- An assignment operator shall return a reference to `*this`.
- An assignment operator shall return a reference to its first arg.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A13-2-3

A relational operator shall return a boolean value.

Description

Rule Definition

A relational operator shall return a boolean value.

Rationale

The return value from relational operators of the C++ Standard Library can be directly checked to see if a relation is true or false. Overloads of the relational operator must be consistent with this usage. Otherwise, users of the overloaded relational operator might see unexpected results. See example below.

Polyspace Implementation

The checker flags overloads of relational operators that do not return a value of type `bool`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Relational Operator Not Returning Boolean Value

```
class aClass {
    int val;
public:
    aClass(int initVal) {
        val = initVal;
    }
    bool operator<=(aClass const& comparingObj )  noexcept{ //Compliant
        return(this->val <= comparingObj.val);
    }
    int operator>=(aClass const& comparingObj ) noexcept { //Noncompliant
        return(this->val <= comparingObj.val? -1:1);
    }
};

void func() {
    aClass anObj(0), anotherObj(1);
    if(anObj <= anotherObj) {
        /* Do something */
    }
    if(anObj >= anotherObj) {
        /* Do something else */
    }
}
```

In this example, the overload of `operator<=` returns a boolean value but the overload of `operator>=` does not return a boolean value. However, in function `func`, the operators `<=` and `>=`

are used as if a boolean value is returned from the overloaded operators. Because the overload of `operator>=` does not return the value zero, the second `if` statement is always true, a result that you might not expect.

Check Information

Group: Overloading

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A13-5-1

If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented.

Description

Rule Definition

If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented.

Rationale

Typically, you overload the subscript operator `operator[]` to provide read and write access to individual elements of an array or similar structure contained in a class. If you implement a non-`const` overload of `operator[]`, you must also implement a `const` version of this overload. Otherwise, you cannot use `operator[]` to read elements of a `const` object.

This rule allows the implementation of a `const` overload of `operator[]` for read-only access without the corresponding non-`const` overload.

Polyspace Implementation

Polyspace flags the definition of the non-`const` member function if no corresponding `const` version of the member function is implemented.

Examples

`const` Version of non-`const` Member Function not Implemented

```
#include <memory>
#include <iostream>

class MyList
{
private:
    static constexpr std::int32_t maxSize = 10;
    std::int32_t container[maxSize];

public:
    std::int32_t& operator[](std::int32_t index) //compliant, non-const version
    {
        return container[index];
    }
    const std::int32_t& operator[](std::int32_t index) const //compliant, const version
    {
        return container[index];
    }
};

class MyList_nc
{
private:
    static constexpr std::int32_t maxSize = 10;
```

```
    std::int32_t container[maxSize];

public:
    std::int32_t& operator[](std::int32_t index) //non-compliant, non-const version only
    {
        return container[index];
    }

};

void func() noexcept
{
    MyList list;
    list[2] = 3; // Uses non-const version of operator[]
    std::cout << list[2] << std::endl;

    const MyList clist = {};
    std::cout << clist[2] << std::endl; // Uses const version of operator[]


}
```

In this example, the overloads of `operator[]` in class `MyList` are compliant because both the `const` and non-`const` versions of the overload are implemented. In class `MyList_nc`, the member function is not compliant because only the non-`const` version was implemented.

Check Information

Group: Overloading

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A13-5-2

All user-defined conversion operators shall be defined explicit.

Description

Rule Definition

All user-defined conversion operators shall be defined explicit.

Rationale

If you do not define a user-defined conversion operator with the `explicit` specifier, compilers can perform implicit and often unintended type conversions from the class type with possibly unexpected results.

The implicit conversion can occur, for instance, when a function accepts a parameter of a type different from the class type that you pass as argument. For instance, the call to `func` here causes an implicit conversion from type `myClass` to `int`:

```
class myClass {} {
  ...
  operator int() {...}
};
myClass myClassObject;

void func(int) {...}
func(myClassObject);
```

Polyspace Implementation

The checker flags declarations or in-class definitions of user-defined conversion operators that do not use the `explicit` specifier.

For instance, `operator int() {}` can convert variable of the current class type to an `int` variable both implicitly and explicitly but `explicit operator int() {}` can only perform explicit conversions.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Missing `explicit` Keyword on Conversion Operator

```
#include <cstdint>

class MyClass {
public:
    explicit MyClass(int32_t arg): val(arg) {};
    operator int32_t() const { return val; }  //Noncompliant
    explicit operator bool() const {   //Compliant
```

```
        if (val>0) {
          return true;
        }
        return false;
     }
private:
    int32_t val;
};

void useIntVal(int32_t);
void useBoolVal(bool);

void func() {
    MyClass MyClassObject{0};
    useIntVal(MyClassObject);
    useBoolVal(static_cast<bool>(MyClassObject));
}
```

In this example, the conversion operator `operator int32_t()` is not defined with the `explicit` specifier and violates the rule. The conversion operator `operator bool()` is defined explicit and does not violate the rule.

When converting to a `bool` variable, for instance, in the call to `useBoolVal`, the `explicit` keyword in the conversion operator ensures that you have to perform an explicit conversion from the type `MyClass` to `bool`. There is no such requirement when converting to an `int32_t` variable. In the call to `useIntVal`, an implicit conversion is performed.

Check Information

Group: Overloading

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A14-7-2

Template specialization shall be declared in the same file (1) as the primary template (2) as a user-defined type, for which the specialization is declared.

Description

Rule Definition

Template specialization shall be declared in the same file (1) as the primary template (2) as a user-defined type, for which the specialization is declared.

Rationale

Observing this rule avoids situations where the behavior is undefined. For instance, if a compiler sees a partial specialization of a template *after* it has instantiated the template, the behavior is undefined. If you specialize a template in the same file as the template, this situation is less likely to occur.

You can also easily extend compile-time interfaces through specialization since the template and its specialization are in the same file and part of the same translation unit. The same reasoning applies to the requirement that a template specialization must be in the same file as the type for which the template is specialized.

Polyspace Implementation

The checker checks each template specialization and raises a violation if:

- The specialization is not in the same file as the template that is specialized.
- The specialization is not in the same file as the user-defined type for which the template is specialized.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Templates

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A14-8-2

Explicit specializations of function templates shall not be used.

Description

Rule Definition

Explicit specializations of function templates shall not be used.

Rationale

Explicit specialization of function templates can cause unexpected issues with overload resolution in C++. Overload resolution:

- First searches for a non-template, plain-old-function that has a matching parameter list.
- If such a function is not available, overload resolution selects the closest matching function template.
- After a function template is selected, the compiler searches for a suitable specialization of the template.

Specializing a template does not change the order of the overload resolution process, which can result in confusing and unexpected behavior. Consider code snippet:

```
//(a) base template
template<class T> void f( T );

//(b) specialization of (a)
template<> void f<>(int*);
//...

//(c) overloads (a)
template<class T> void f( T* );

//...
main(){
    int *p;
    f( p );
}
```

When `f()` is called with an `int*` in `main()`, you might expect the specialization for `int*`, marked (b), to be called. The compiler resolves the call to `f()` as follows:

1. The compiler searches for a plain-old-function with input type `int*`.
2. Because there is no such function, the compiler searches for a function template that has the closest matching parameter list.
3. The template (c), which takes a generic pointer as input, is the closest match for `f(int*)`.
4. The template (c) is not specialized. The overload resolution process stops and calls the template in (c).

Even though a specialized template for `int*` type input is defined in (b), the overload resolves to the template in (c) instead, which can be unexpected.

When you specialize an overloaded function template, the overload resolution process can get more confusing. Which among the overloaded templates gets specialized depends on the order of declaration. Consider the code snippet:

```
//(a)
template <typename T> void F1(T t){}
//(b)
template <typename T> void F1(T* p){}
//(x): Specialization of template
template <> void F1<>(uint16_t* p){}
```

You cannot determine whether (x) specializes (a) or (b) from the declaration alone. it depends on the declaration order. For instance, in the preceding case (x) specializes (b). But in this case, (x) specializes (a):

```
//(a)
template <typename T> void F1(T t){}
//(x): Specialization of template
template <> void F1<>(uint16_t* p){}
//(b)
template <typename T> void F1(T* p){}
```

To avoid confusing code and unexpected behavior, avoid specializing function templates. If you must specialize a function template, then write a single function template that delegates to a class template. For example, in this code, a function template `f()` delegates to the class `f_implementation`.

```
template<class T> class f_implementation;

template<class T> void f( T t ) {
    FImpl<T>::f( t ); //Don't specialize function template
}

template<class T> class f_implementation {
    static void f( T t ); // Specializing class templates is permissible.
}
```

Delegating to a class template also enables partial specialization.

Polyspace Implementation

If you explicitly specialize a function template, Polyspace flags the function template.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Avoid Specializing Function Templates

This example shows how Polyspace flags specialized function templates.

```
#include <cstdint>
#include <memory>
#include <iostream>
//(a)
```

```
template <typename T> void F1(T t){
  std::cout << "(a)" << std::endl;
}
//(x) specializes (a)
template <> void F1<>(uint16_t* p){// Noncompliant
  std::cout << "(x)" << std::endl;
}
//(b) overloads (a)
template <typename T> void F1(T* p){// Compliant
  std::cout << "(b)" << std::endl;
}
//(y) specializes (b)
template <> void F1<>(uint8_t* p){// Noncompliant
  std::cout << "(c)" << std::endl;
}
//(d) plain old function overloads (a) and (b)
void F1(uint8_t* p){                    // Compliant
  std::cout << "(d)" << std::endl;
}

int main(void)
{
    auto sp8 = std::make_unique<uint8_t>(3);
    auto sp16 = std::make_unique<uint16_t>(3);
    F1(sp8.get()); //calls (d), might expect (y)
    F1(sp16.get()); //calls (b), might expect (x)
    return 0;
}
```

When the function `F1()` is called in main, overload resolution determines which instance of `F1()` is called.

- When F1() is invoked with pointers to `uint8_t`, the compiler calls the plain-old-function (d) because it takes precedence. You might incorrectly expect the specialization (y) to be called.
- When F1() is invoked with pointers to `uint16_t`, the compiler calls the overloaded template (b) because it is the closest matching template. You might incorrectly expect the specialization (x) to be called.

Specializing function templates can cause confusion and unexpected results. Polyspace flags the specialized function templates (x) and (y).

Check Information

Group: Templates

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A15-1-2

An exception object shall not be a pointer.

Description

Rule Definition

An exception object shall not be a pointer.

Polyspace Implementation

The checker raises a violation if a `throw` statement throws an exception of pointer type.

The checker does not raise a violation if a NULL pointer is thrown as exception. Throwing a NULL pointer is forbidden by `AUTOSAR C++14 Rule M15-1-2`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A15-3-5

A class type exception shall be caught by reference or const reference.

Description

Rule Definition

A class type exception shall be caught by reference or const reference.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A15-5-3

The std::terminate() function shall not be called implicitly.

Description

Rule Definition

The std::terminate() function shall not be called implicitly.

Polyspace Implementation

The checker flags these situations when the terminate() function can be called implicitly:

- An exception escapes uncaught. For instance:
  - Before an exception is caught, it escapes through another function that throws an uncaught exception. For instance, a catch statement or exception handler invokes a copy constructor that throws an uncaught exception.
  - A throw expression with no operand rethrows an uncaught exception.
- A class destructor throws an exception.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A16-0-1

The preprocessor shall only be used for unconditional and conditional file inclusion and include guards, and using specific directives.

Description

Rule Definition

The preprocessor shall only be used for unconditional and conditional file inclusion and include guards, and using specific directives.

Rationale

Other than unconditional and conditional file inclusion and include guards, avoid the use of preprocessor directives. Use a safer alternative instead. For instance:

- Instead of:

  ```
  #define MIN(a,b) ((a < b)? (a) : (b))
  ```

  You can use inline functions and function templates.
- Instead of:

  ```
  #define MAX_ARRAY_SIZE 1024U
  ```

  You can use a constant object.

In these situations, preprocessor directives do not provide the benefits that the alternatives provide, such as linkage, type checking, overloading, and so on.

Polyspace Implementation

The rule checker does not allow the use of preprocessor directives. The only exceptions are:

- `#ifdef`, `#ifndef`, `#if`, `#if defined`, `#elif`, `#else` and `#endif`, only if used for conditional file inclusion and include guards.
- `#define` only if used for defining macros to be used in include guards. For instance, in this example, the macro `__FILE_H__` prevents the contents of the header file from being included more than once:

  ```
  /* aHeader.h */

  #ifndef __FILE_H__
  #define __FILE_H__
     /* Contents of header file */
  #endif
  ```

  When `#ifdef`, `#define` and `#endif` are used as include guards in a header file, the entire content of the header file must be in the include guard.
- `#include`

The checker does not allow the `#define` directives in other contexts. If you use `#define`-s for purposes other than for include guards, do one of the following:

- To define macros when compiling your code, instead of `#define`-s, use compilation flags (such as the GCC option `-D`). When running a Polyspace analysis, use the equivalent Polyspace option `Preprocessor definitions (-D)`.
- To retain the use of `#define` in your code, justify the violation using comments in your results or code. See "Address Polyspace Results Through Bug Fixes or Justifications".

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Compliant and Noncompliant Use of Preprocessor Directives

```
#include <cstdint>       //Compliant: unconditional file inclusion

#ifdef WIN32             //Compliant: include guard
   #include <windows.h>  //Compliant: conditional file inclusion
#endif

#ifdef WIN32     //Noncompliant
    std::int32_t func(std::int16_t x, std::int16_t y) noexcept;
#endif
```

In this example, the rule is not violated when preprocessor directives are used for unconditional and conditional inclusion and include guards. Otherwise, the rule is violated.

Check Information

Group: Preprocessing directives

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

"Check for Coding Standard Violations"

Introduced in R2019b

AUTOSAR C++14 Rule A16-2-1

The ', ", /*, //, \ characters shall not occur in a header file name or in #include directive.

Description

Rule Definition

The ', ", /, //, \ characters shall not occur in a header file name or in #include directive.*

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A16-6-1

#error directive shall not be used.

Description

Rule Definition

#error directive shall not be used.

Rationale

You typically use the `#error` directive by combining it with a `#if` or similar directive to make the compilation fail and issue a message when a condition is not met. However, you cannot apply `#error` to templates. Preprocessor directives do not obey linkage, type checker, overloading and other C++ features, and `#error` will not be evaluated as a per-instance template deduction.

Instead, use `static_assert` for compile-time error checking. Static assertions provide all the benefits of C++ features and make the code clearer.

Polyspace Implementation

Polyspace flags all uses of the `#error` directive.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing directives

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++ 14 Rule A16-7-1

The #pragma directive shall not be used.

Description

Rule Definition

The #pragma directive shall not be used.

Rationale

The use of the `#pragma` directive in your code results in implementation-defined behavior. The directive might also not be supported by certain compilers.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `#pragma once` Directive

```
//header.h
#pragma once //Noncompliant

#ifndef HEADER_H_ //Compliant
#define HEADER_H_
// ...
// body of header file
//..
#endif
```

The `#pragma once` directive prevents the inclusion of `header.h` more than once. However, if you copy `header.h` into multiple project modules, the directive may or may not treat the copies as the same file depending on the implementation. To avoid double definitions, use the `#ifndef` include guard instead.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019b

AUTOSAR C++14 Rule A17-0-1

Reserved identifiers, macros and functions in the C++ standard library shall not be defined, redefined or undefined.

Description

Rule Definition

Reserved identifiers, macros and functions in the C++ standard library shall not be defined, redefined or undefined.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Library Introduction

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A17-6-1

Non-standard entities shall not be added to standard namespaces.

Description

Rule Definition

Non-standard entities shall not be added to standard namespaces.

Rationale

Adding declarations or definitions to namespace `std` or its subspaces:or to `posix` or its subspaces, leads to undefined behavior. For instance, any addition within braces here leads to undefined behavior:

```
namespace std {
  ...
}
```

Likewise, explicitly specializing a member function or member class of a standard library leads to undefined behavior.

Polyspace Implementation

The checker flags additions to the namespaces `std`, `posix`, or their subspaces, or specializations of class or function templates from these namespaces.

The rule specification allows exceptions to the specialization aspect of the rule for standard library templates that require a user-defined type. If you have a process that all rule violations must be justified and an issue flagged by the checker belongs to this category of exceptions, justify the issue using comments in your result or code. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Library introduction

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A18-0-1

The C library facilities shall only be accessed through C++ library headers.

Description

Rule Definition

The C library facilities shall only be accessed through C++ library headers.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language Support Library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A18-0-2

The error state of a conversion from string to a numeric value shall be checked.

Description

Rule Definition

The error state of a conversion from string to a numeric value shall be checked.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language Support Library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A18-0-3

The library <clocale> (locale.h) and the setlocale function shall not be used.

Description

Rule Definition

The library <clocale> (locale.h) and the setlocale function shall not be used.

Polyspace Implementation

`setlocale` and `localeconv` should not be used as a macro or a global with external "C" linkage.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++ 14 Rule A18-1-1

C-style arrays shall not be used

Description

Rule Definition

C-style arrays shall not be used.

Rationale

A C-style array is an array that is not wrapped in a class such as `std::array` when the array is declared. You can lose information about the size of a C-style array. For instance, an array that you pass to a function decays to a pointer to the first element of the array. This can lead to unsafe and difficult to maintain code.

The AUTOSAR standard allows declarations of `static constexpr` data members of a C-style array type. For example, this declaration is compliant.

```
class A
{
  public:
    static constexpr std::uint8_t array[] {0, 1, 2}; // Compliant by exception
};
```

Polyspace Implementation

The rule checker does not flag C-style array arguments in function declarations because the rule violation still exists if you fix the function declaration and not the definition. A function might be declared in your code and defined in a library that you cannot access. The checker flags C-style array arguments in function definitions. For instance, in this code snippet, the checker flags the argument of `foo` but not the argument of `bar`.

```
extern void bar(char arg[]); //Declaration, checker raises no rule violation
int foo(char arg[]) // Definition, checker raises a rule violation
{
    return sizeof(arg); //Returns size of pointer, not size of array
}
void baz()
{
    char value[10]; //C-style array, checker raises a rule violation
    assert(sizeof(value) == foo(value));
}
```

The checker raises a flag on `arg` in the definition of `foo` even when there is no explicit C-style array definition for the argument. For example, declaring `char* value;` instead of `char value[10];` in `baz()` would still result in a rule violation on the argument of `foo`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Declaration of C-Style Array

```
#include <array>

void func()
{

    const std::uint8_t size = 10;
    std::int32_t a1[size]; //non-compliant
    std::array<std::int32_t, size> a2; //compliant

}
```

In this example, the rule is violated when you declare C-style array `a1`. To declare fixed-size stack-allocated arrays, use `std:array` instead.

Check Information

Group: 18 Language Support Library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019b

AUTOSAR C++ 14 Rule A18-1-2

The std::vector<bool> specialization shall not be used.

Description

Rule Definition

The std::vector<bool> specialization shall not be used.

Rationale

The specialization of `std::vector` for the type `bool` can be made space-efficient in an implementation defined manner. For instance, `std::vector<bool>` does not necessarily store its elements as a contiguous array. As a result, the specialization does not work as expected with all standard library template (STL) algorithms, such as the index `operator[]()` which does not return a contiguous sequence of elements. You cannot safely modify distinct elements of STL container `std::vector<bool>`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Non-Compliant and Compliant Use of `std::vector` With `bool` Type

```
#include <cstdint>
#include <vector>

class BoolWrapper
{
public:
    BoolWrapper() = default;
    constexpr BoolWrapper(bool b) : b_(b) {}
    constexpr operator bool() const
    {
        return b_;
    }
private:
    bool b_{};
};

void Fn() noexcept
{
    std::vector<bool> v2; //non-compliant
    std::vector<BoolWrapper> v3{true, false, true, false}; //compliant
}
```

In this example, vector `v2` is non-compliant because it is declared with `std::vector<bool>`. A possible fix is to use `std::vector` with a value type `BoolWrapper` that wraps `bool`.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019b

AUTOSAR C++14 Rule A18-1-3

The std::auto_ptr shall not be used.

Description

Rule Definition

The std::auto_ptr shall not be used.

Rationale

The `std::auto_ptr` is a type of class template that predates the introduction of move semantics in the C++11 language standard. When you copy a source `std::auto_ptr` object into a target object, the source object is modified. The compiler transfers the ownership of the resources in the source object to the target object and sets the source object to a null-pointer. Because of this unusual copy syntax, using the source object after the copy operation might lead to unexpected behavior. Consider this code snippet where the use of `std::auto_ptr` results in a segmentation fault.

```
void func(auto_ptr<int> p) {
    cout<<*p;
    //...
}

int main()
{
    std::auto_ptr<int> s = new int(1);
    //..
    func(s); // This call makes s a null-pointer
    //...
    func(s); // exception, because s is null
    return 1;
}
```

The first call to `func()` copies the source `std::auto_ptr` object `s` to the argument `p`, transfers ownership of the pointer to `p`, and sets `s` to a null pointer. When `func()` is called again, the compiler tries to access the null-pointer `s`, causing a segmentation fault.

The `std::auto_ptr` type objects are also incompatible with any generic code that expects a copy operation to not invalidate the source object, such as the standard template library (STL). Avoid using `std::auto_ptr`. It is deprecated in C++11 and removed from C++17. The C++11 language standard introduces `std::unique_ptr` as a safer replacement for `std::auto_ptr`. Use `std::unique_ptr` instead of `std::auto_ptr`.

Polyspace Implementation

Polyspace flags all instances of `std::auto_ptr` in your code, other than those in C style arrays.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Avoid Using `std::auto_ptr`

This code shows how Polyspace flags `std::auto_ptr` in your code.

```
#include <cstdint>
#include <memory>
#include <vector>
#define AUTOPTROF(_TYPE) std::auto_ptr<_TYPE>
AUTOPTROF(int) v_int;                           // Noncompliant
typedef struct {
    std::auto_ptr<bool> vb;                     // Noncompliant
} T;
T vec;
typedef std::auto_ptr<int> my_int_auto_ptr;     // Noncompliant
void Fn() noexcept
{
      // Noncompliant
    std::auto_ptr<std::int32_t> ptr1(new std::int32_t(10));
    std::unique_ptr<std::int32_t> ptr2 =
    std::make_unique<std::int32_t>(10);         // Compliant
    std::vector<std::auto_ptr<std::int32_t>> v;  // Noncompliant
}

int main(){
    //..
}
```

Polyspace flags the `std::auto_ptr` objects. Use `std::unique_ptr` instead of `std::auto_ptr`.

Check Information

Group: Language support library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A18-1-6

All std::hash specializations for user-defined types shall have a noexcept function call operator.

Description

Rule Definition

All std::hash specializations for user-defined types shall have a noexcept function call operator.

Rationale

`std::hash` specializations provided by the standard library have a guarantee of no exceptions. If you manually create a `std::hash` specialization, emulate this guarantee for your specialization. Define all specializations of `std::hash` for your custom data types as `noexcept`.

Otherwise, standard library containers that use your specialization of `std::hash` indirectly might throw uncaught exceptions. The exceptions are not caught because the standard library containers do not provide a way to use `try-catch` blocks for exceptions from `std::hash`.

Polyspace Implementation

The checker flags specializations of the `std::hash` template with user defined types that do not have a `noexcept` specifier.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language support library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++ 14 Rule A18-5-1

Functions malloc, calloc, realloc and free shall not be used.

Description

Rule Definition

Functions malloc, calloc, realloc and free shall not be used.

Rationale

C-style memory allocation and deallocation using `malloc`, `calloc`, `realloc`, or `free` is not type safe and does not invoke class's constructors/destructor to create/delete objects.

For instance, `malloc` allocates memory to an object and returns a pointer to the allocated memory of type `void*`. A program can then implicitly cast the returned pointer to a different type that might not match the intended type of the object.

The use of these allocation and deallocation functions can result in undefined behavior if:

- You use `free` to deallocate memory allocated with operator `new`.
- You use operator `delete` to deallocate memory allocated with `malloc`, `calloc`, or `realloc`.

The rule is not violated when you perform dynamic memory allocation or deallocation using overloaded `new` and `delete` operators, or custom implementations of `malloc` and `free`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Non-Compliant Use of `malloc`

```
#include <cstdint>
#include <cstdlib>

void func()
{

    std::int32_t* p1 = static_cast<std::int32_t*>(malloc(sizeof(std::int32_t))); // Non-compliant
    *p1 = 0;

    free(p1); // Non-compliant

    std::int32_t* p2 = new std::int32_t(0); // Compliant

    delete p2; // Compliant
}
```

In this example, the allocation of memory for pointer `p1` using `malloc` and the memory deallocation using `free` are non-compliant. These operations are not type safe. Instead, use operators `new` and `delete` to allocate and deallocate memory.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019b

AUTOSAR C++14 Rule A18-5-2

Operators new and delete shall not be called explicitly.

Description

Rule Definition

Operators new and delete shall not be called explicitly.

Rationale

The `new` operator allocates memory. The `delete` operator deallocates memory. If you use these operators explicitly, you must couple them to avoid memory leak. Even then, unexpected exceptions or returns can result in memory leak. Consider this code where memory is allocated for a pointer by explicitly calling `new` and deallocated by explicitly calling `delete`.

```
std::int32_t ThrowError(){
    std::int32_t errorCode;
    std::int31_t* ptr = new std::int32_t{0};
    //...
    if(errorCode!=0){
        throw std::runtime_error{"Error"};
    }
    //...
    if (errorCode != -1) {
        return 1;
    }
    delete ptr;
    return errorCode;
}
```

Even though the `new` operator is coupled with a `delete` operator, this code can lead to memory leak in certain conditions.

- If the first `if()` statement is `true`, then the function produces an exception and exits without deleting the pointer.
- If the second `if()` statement is `true`, then the function returns `1` and exits without deleting the pointer.

To avoid an unpredictable memory leak, encapsulate resources, such as dynamically allocated memory or file handles, in objects. Acquire the resources in object constructors and release the resources in object destructors. This design pattern is called "Resource Acquisition Is Initialization" or RAII. Following the RAII pattern prevents a memory leak even when there are unexpected exceptions and returns.

Alternatively, use manager objects that manage the lifetime of dynamically allocated resources. Examples of manager objects in the standard library include:

- `std::unique_ptr` along with `std::make_unique`
- `std::shared_ptr` along with `std::make_shared`
- `std::string`

- `std::vector`

This rule does not apply to a `new` operator or a `delete` operator in user-defined RAII classes and managers.

Polyspace Implementation

AUTOSAR C++14 permits explicit resource allocation by calling the `new` operator in two cases, when the allocated resource is immediately passed to:

- A manager object
- A RAII class that does not have a safe alternative to the `new` operator.

Polyspace flags all explicit uses of the `new` operator and the `delete` operator. If you have a process where a `new` operator can be permissible and there is no safer alternative, justify the issue by using comments in your result or code. See "Address Polyspace Results Through Bug Fixes or Justifications".

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Avoid Explicitly Calling `new` Operator and `delete` Operator

This code shows how Polyspace flags `new` or `delete` operators.

```
#include <cstdint>
#include <memory>
#include <vector>
#include <cstddef>

using namespace std;

int32_t Fn1()
{
    int32_t errorCode{0};
    int32_t* ptr =
    new int32_t{0}; //Noncompliant
    // ...
    if (errorCode != 0) {
        throw  runtime_error{"Error"}; // Possible Memory Leak
    }
    // ...
    if (errorCode != 0) {
        return 1; //Possible Memory Leak
    }
    // ...
    delete ptr; //Noncompliant

    return errorCode; // Possible Memory Leak
}

int32_t Fn2()
{
```

```
    int32_t errorCode{0};
    // Alternative to 'new'
    unique_ptr<int32_t> ptr1 = make_unique< int32_t>(0);
    unique_ptr<int32_t> ptr2(new  int32_t{0});   // Noncompliant
    shared_ptr<int32_t> ptr3 =
    make_shared<int32_t>(0);   //Compliant
    vector<int32_t> array;   // Compliant

    if (errorCode != 0) {
        throw  runtime_error{"Error"};   // No memory leaks
    }
    // ...
    if (errorCode != 0) {
        return 1; // No memory leaks
    }
    // ...
    return errorCode; // No memory leaks
}


class X
{
public:
    static void* operator new( size_t s)
    {
        return ::operator new(s);   // Noncompliant
    }

    static void* operator new[]( size_t s)
    {
        return ::operator new(s);   // Noncompliant
    }

    static void operator delete(void* ptr,  size_t s)
    {
        ::operator delete(ptr);   // Noncompliant
    }

    static void operator delete[](void* ptr,  size_t s)
    {
        ::operator delete(ptr);   // Noncompliant
    }
};

main(){
    X* x1    = new X;   // Noncompliant
    X* x2    = new X[2];   // Noncompliant
}
```

In `Fn1()`, the operators `new` and `delete` are explicitly called for resource management. Consequently, an unexpected exception or return can lead to a memory leak. Polyspace flags the `new` and `delete` operators. In `Fn2()`, manager objects are used for memory management. Even in cases of unexpected exceptions and returns, there are no memory leaks in `Fn2()`.

The class `X` contains custom overloads for `new` and `delete` operators. Polyspace flags all instances of `new` and `delete` operators in the definitions of the custom overloads. In main(), Polyspace also flags the overloaded new and delete operators.

Check Information

Group: Language support library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A18-5-3

The form of delete operator shall match the form of new operator used to allocate the memory.

Description

Rule Definition

The form of delete operator shall match the form of new operator used to allocate the memory..

Rationale

- The `delete` operator releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.
- If you use the single-object notation for `delete` on a pointer that is previously allocated with the array notation for `new`, the behavior is undefined.

The issue can also highlight other coding errors. For instance, you perhaps wanted to use the `delete` operator or a previous `new` operator on a different pointer.

Polyspace Implementation

The checker flags a defect when:

- You release a block of memory with the `delete` operator but the memory was previously not allocated with the `new` operator.
- You release a block of memory with the `delete` operator using the single-object notation but the memory was previously allocated as an array with the `new` operator.

This defect applies only to C++ source files.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Deleting Static Memory

```
void assign_ones(void)
{
    int ptr[10];

    for(int i=0;i<10;i++)
        *(ptr+i)=1;

    delete[] ptr;
}
```

The pointer `ptr` is released using the `delete` operator. However, `ptr` points to a memory location that was not dynamically allocated.

Correction: Remove Pointer Deallocation

If the number of elements of the array `ptr` is known at compile time, one possible correction is to remove the deallocation of the pointer `ptr`.

```
void assign_ones(void)
{
    int ptr[10];

    for(int i=0;i<10;i++)
        *(ptr+i)=1;
}
```

Correction — Add Pointer Allocation

If the number of array elements is not known at compile time, one possible correction is to dynamically allocate memory to the array `ptr` using the `new` operator.

```
void assign_ones(int num)
{
    int *ptr = new int[num];

    for(int i=0; i < num; i++)
        *(ptr+i) = 1;

    delete[] ptr;
   }
```

Mismatched `new` and `delete`

```
int main (void)
{
    int *p_scale = new int[5];

    //more code using scal

    delete p_scale;
}
```

In this example, `p_scale` is initialized to an array of size 5 using `new int[5]`. However, `p_scale` is deleted with `delete` instead of `delete[]`. The `new`-`delete` pair does not match. Do not use `delete` without the brackets when deleting arrays.

Correction — Match `delete` to `new`

One possible correction is to add brackets so the `delete` matches the `new []` declaration.

```
int main (void)
{
    int *p_scale = new int[5];

    //more code using p_scale

    delete[] p_scale;
}
```

Correction — Match `new` to `delete`

Another possible correction is to change the declaration of `p_scale`. If you meant to initialize `p_scale` as 5 itself instead of an array of size 5, you must use different syntax. For this correction, change the square brackets in the initialization to parentheses. Leave the `delete` statement as it is.

```
int main (void)
{
    int *p_scale = new int(5);

    //more code using p_scale

    delete p_scale;
}
```

Check Information

Group: Language Support Library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A18-5-4

If a project has sized or unsized version of operator 'delete' globally defined, then both sized and unsized versions shall be defined.

Description

Rule Definition

If a project has sized or unsized version of operator 'delete' globally defined, then both sized and unsized versions shall be defined.

Rationale

The C++14 Standard defines a sized version of `operator delete`. For instance, for an unsized `operator delete` with this signature:

```
void operator delete (void* ptr);
```

The sized version has an additional size argument:

```
void operator delete (void* ptr, std::size_t size);
```

See the C++ reference page for `operator delete`.

The Standard states that if both versions of `operator delete` exist, the sized version must be called because it provides a more efficient way to deallocate memory. However, in some cases, for instance to delete incomplete types, the unsized version is used.

If you overload the unsized version of `operator delete`, you must also overload the sized version. You typically overload `operator delete` to perform some bookkeeping in addition to deallocating memory on the free store. If you overload the unsized version but not the sized one or the other way around, any bookkeeping you perform in one version will be omitted from the other version. This omission can lead to unexpected results.

Polyspace Implementation

The checker flags situations where an unsized version of `operator delete` exists but the corresponding sized version is not defined, or vice versa.

The checker is enabled only if you specify a C++ version of C++14 or later. See `C++ standard version (-cpp-version)`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Missing Sized Overload of `operator delete[]`

```
#include <new>
#include <cstdlib>
```

```
int global_store;

void update_bookkeeping(void *allocated_ptr, bool alloc) {
   if(alloc)
      global_store++;
   else
      global_store--;
}

void operator delete(void *ptr);
void operator delete(void* ptr) {
    update_bookkeeping(ptr, false);
    free(ptr);
}

void operator delete(void *ptr, std::size_t size);
void operator delete(void* ptr, std::size_t size) {
    //Compliant, both sized and unsized version defined
    update_bookkeeping(ptr, false);
    free(ptr);
}

void operator delete[](void *ptr);
void operator delete[](void* ptr) { //Noncompliant, only unsized version defined
    update_bookkeeping(ptr, false);
    free(ptr);
}
```

In this example, both the unsized and sized version of `operator delete` are overloaded and complies with the rule. However, only the unsized version of `operator delete[]` is overloaded, which violates the rule..

Check Information

Group: Language Support Library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)` | `Invalid deletion of pointer` | `Invalid free of pointer` | `Memory leak` | `Mismatched alloc/dealloc functions on Windows` | `Missing overload of allocation or deallocation function`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++ 14 Rule A18-9-1

The std::bind shall not be used.

Description

Rule Definition

The std::bind shall not be used.

Rationale

`std::bind` takes a callable object, such as a function object, and produces a forwarding call wrapper for this object. Calling the wrapper invokes the object with some of the object arguments bound to arguments you specify in the wrapper. For instance, in this code snippet, `foo` is called through `bar` with the first (second) argument of `bar` bound to the second (first) argument of `foo`.

```
int foo(int, int);
auto bar = std::bind(foo, _2, _1);
bar(10, 20); //call to foo(20, 10)
```

The use of `std::bind` results in a less readable function call. A developer that is unfamiliar with `foo` would need to see the declaration of `foo` to understand how to pass arguments to `bar`, and might confuse one function parameter with another. In addition, a compiler is less likely to inline a function that you create using `std::bind`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Non-Compliant Use of `std::bind`

```
#include <cstdint>polys
#include <functional>
class A
{
//...
};
void func(A const& a, double y) noexcept
{
//...
}
void func1() noexcept
{
    double arg2 = 0.0;
    auto bind_fn = std::bind(&func, std::placeholders::_1, arg2); // Non-compliant
    // ...
    A const a{};
    bind_fn(a);
}
void func2() noexcept
{
    auto lambda_fn = [](A const & a) -> void { // Compliant
        double arg2 = 0.0;
        func(a, arg2);
    }; // Compliant
    // ...
    A const a{};
    lambda_fn(a);
}
```

In this example, `func` is called through `bind_fn` with the only argument of `bind_fn` bound to the first argument of `func`. It might be unclear to a developer that `arg2` in the definition of `bind_fn` is the second argument of `func`. For a more readable code, use lambda expressions instead. The call to `func` with two arguments is clearer in the definition of `lambda_fn`.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019b

AUTOSAR C++14 Rule A18-9-3

The std::move shall not be used on objects declared const or const&.

Description

Rule Definition

The std::move shall not be used on objects declared const or const&.

Rationale

When you use `std::move()` on an object, it is cast into an rvalue. The compiler then manages the resources in the object by calling the constructor or operator with the closest matching parameter list. If you call `std::move()` on a `const` or `const&` type object, the call returns a `const` or `const&` type rvalue. Because move constructors and operators do not take a `const` type argument, the compiler calls the copy constructor or operator instead of the move constructor or operator. Consider this code snippet where a `const` object is copied when you might expect a move after a call to `std::move()`.

```
class string{
    //...
public:
    string(const string& rhs);// copy contructor
    string(string&& rhs);     //move constructor
};

void print(string text) {
    cout<<text;
    //...
}

int main(){
    int const message = "Error";
    //..
    print(std::move(message))// the copy constructor is called
}
```

The return type of `std::move(message)` is the rvalue `const string&&`. Between the move and copy constructors of class `string`, only the copy constructor accepts `const` type argument. The compiler calls the copy constructor and copies the resources of `message` into `text`.

Because `std::move()` does not move a `const` or `const&` type object, avoid using `std::move()` on `const` or `const&` objects. If you intend to move resources from an object, do not declare it as `const` or `const&`.

Polyspace Implementation

Polyspace flags use of `std::move()` on:

- Objects that are declared `const` or `const&`.
- Objects that are cast to `const` or `const&`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Avoid Using `std::move()` on `const` and `const&` Objects

```
#include <cstdint>
#include <utility>
class A
{
    // Implementation
};
void F1(const int32_t &is_const, int32_t &is_non_const)
{
    const A a1{};
    int32_t target = 0;
    A a2 = a1;              // Compliant
    A a3 = std::move(a1);   // Noncompliant

    target =
    std::move((const int32_t &)is_non_const);// Noncompliant
    target =
    std::move(static_cast<const int32_t &>(is_non_const));// Noncompliant
    target =
    std::move(const_cast<int32_t &>(is_const));// Compliant
}
int main(){
    //...
}
```

- Polyspace flags the use of `std::move()` with `const` object `a1`. The compiler calls the copy constructor to copy `a1` to `a3`. You might expect the compiler to call the move constructor.
- Polyspace also flags the use of `std::move()` with the object `is_non_const` when it is cast to `const`. After the casting, the compiler calls the copy constructor to copy `is_non_const` to `target`. You might expect the compiler to call the move constructor.
- Polyspace does not flag the use of `std::move()`with the non-`const` object that results from casting the `const` object `is_const` into a non-`const` type by using `const_cast`. After casting, `is_const` is no longer a `const` object. The compiler calls the move constructor.

Check Information

Group: Language support library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A2-10-1

An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

Description

Rule Definition

An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.

Rationale

The rule flags situations where the same identifier name is used in two variable declarations, one in an outer scope and the other in an inner scope.

```
int var;
...
{
...
  int var;
...
}
```

All uses of the name in the inner scope refers to the variable declared in the inner scope. However, a developer or code reviewer can incorrectly assume that the usage refers to the variable declared in the outer scope. In all cases flagged by this rule, you cannot clarify the usage further using the scope resolution operator.

Polyspace Implementation

The rule checker flags all cases of variable shadowing except when:

- The same identifier name is used in an outer and inner named namespace.
- The same name is used for a class data member and a variable outside the class.
- The same name is used for a method in a base and derived class.

The checker flags even those cases where the variable declaration in the outer scope occurs *after* the variable declaration in the inner scope. In those cases, though the variable hiding does not occur, reusing the variable name can cause developer confusion.

The rule does not flag these situations because you can clarify whether an usage of the variable refers to the variable in the inner or outer scope. For instance, in this example:

```
int var;

namespace n1 {
   int var;
}
```

within the namespace `n1`, you can refer to the variable in the inner scope as `n1::var` and the global variable as `::var`.

The rule checker also does not detect these issues:

- A variable in an unnamed namespace hides another variable in an outer scope.
- A variable local to a lambda expression hides a captured variable.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Local Variable Hiding Global Variable

```
int varInit = 1;

void doSomething(void);

void step(void) {
    int varInit = 0; //Noncompliant
    if(varInit)
       doSomething();
}
```

In this example, `varInit` defined in `func` hides the global variable `varInit`. The `if` condition refers to the local `varInit` and the block is unreachable, but you might expect otherwise.

Loop Index Hiding Variable Outside Loop

```
void runSomeCheck(int);

void checkMatrix(int dim1, int dim2) {
  for(int index = 0; index < dim1; index++) {
      for(int index = 0; index < dim2; index++) {
          runSomeCheck(index);
      }
  }
}
```

In this example, the variable `index` defined in the inner `for` loop hides the variable with the same name in the outer loop.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A2-11-1

Volatile keyword shall not be used.

Description

Rule Definition

Volatile keyword shall not be used.

Polyspace Implementation

Reports if volatile keyword is used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A2-13-1

Only those escape sequences that are defined in ISO/IEC 14882:2014 shall be used.

Description

Rule Definition

Only those escape sequences that are defined in ISO/IEC 14882:2014 shall be used.

Rationale

Escape sequences are certain special characters represented in string and character literals. They are written with a backslash (\) followed by a character.

The C++ Standard (ISO/IEC 14882:2003, Sec. 2.13.2) defines a list of escape sequences. See Escape Sequences. Use of escape sequences (backslash followed by character) outside that list leads to undefined behavior.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Incorrect Escape Sequences

```
void func () {
  const char a[2] = "\k"; \\Noncompliant
  const char b[2] = "\b"; \\Compliant
}
```

In this example, `\k` is not a recognized escape sequence.

Check Information

Group: Lexical Conventions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A2-13-2

String literals with different encoding prefixes shall not be concatenated.

Description

Rule Definition

String literals with different encoding prefixes shall not be concatenated.

Rationale

Narrow string literals are enclosed in double quotes without a prefix. Wide string literals are enclosed in double quotes with a prefix `L` outside the quotes. See string literals.

Concatenation of narrow and wide string literals can lead to undefined behavior.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Concatenation of Narrow and Wide String Literals

```
char array[] = "Hello" "World";
wchar_t w_array[] = L"Hello" L"World";
wchar_t mixed[] = "Hello" L"World"; //Noncompliant
```

In this example, in the initialization of the array `mixed`, the narrow string literal `"Hello"` is concatenated with the wide string literal `L"World"`.

Check Information

Group: Lexical Conventions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A2-13-3

Type wchar_t shall not be used

Description

Rule Definition

Type wchar_t shall not be used.

Rationale

The size of `wchar_t` is implementation-dependent. If you use `wchar_t` for Unicode values, your code is bound to a specific compiler.

To improve the portability of your code, use `char16_t` and `char32_t` instead. These are standard types introduced in C++11 for text strings with UTF-16 and UTF-32 encodings.

Polyspace Implementation

The rule checker flags all uses of the `wchar_t` keyword.

If you do not want to fix the issue, add a comment justifying the result. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `wchar_t` Keyword

```
char16_t str1[] = u"A UTF-16 string";  //Compliant
char32_t str2[] = U"A UTF-32 string";  //Compliant
wchar_t str3[] = L"A Unicode string";  //Noncompliant
```

The use of `wchar_t` violates this rule. Instead the types `char16_t` and `char32_t` can be used.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A2-13-4

String literals shall not be assigned to non-constant pointers

Description

Rule Definition

String literals shall not be assigned to non-constant pointers.

Rationale

This rule prevents assignments of string literals to pointers that point to non `const` objects. Such assignments allow later modification of the string literal.

An attempt to modify a string literal can result in undefined behavior. For example, some implementations can store string literals in read-only memory. An attempt to modify the string literal can result in an exception or crash.

Later C++ standards require a compiler warning for such modifications. The rule is in place for situations when you suppress compiler warnings (and AUTOSAR C++14 rules associated with those warnings).

Polyspace Implementation

The rule checker flags assignment of string literals to pointers other than pointers to `const` objects.

The checker does not flag assignment of string literals to non-`const` arrays. The checker for `AUTOSAR C++ 14 Rule A18-1-1` forbids direct use of C-style arrays and prevents these assignments.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Incorrect Assignment of String Literal

```
char *str1 = "xxxxxx";            /* Non-Compliant */
const char *str2 = "xxxxxx";      /* Compliant */

void checkSystem1(char*);
void checkSystem2(const char*);

void main() {
 checkSystem1("xxxxxx");    /* Non-Compliant */
 checkSystem2("xxxxxx");    /* Compliant */
}
```

In this example, the rule is not violated when string literals are assigned to `const char*` pointers, either directly or through copy of function arguments. The rule is violated only when the `const` qualifier is not used.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A2-13-5

Hexadecimal constants should be upper case.

Description

Rule Definition

Hexadecimal constants should be upper case.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A21-8-1

Arguments to character-handling functions shall be representable as an unsigned char.

Description

Rule Definition

Arguments to character-handling functions shall be representable as an unsigned char.

Rationale

Comparison with EOF: Suppose, your compiler implements the plain `char` type as signed. In this implementation, the character with the decimal form of 255 (–1 in two's complement form) is stored as a signed value. When you convert a `char` variable to the wider data type `int` for instance, the sign bit is preserved (sign extension). This sign extension results in the character with the decimal form 255 being converted to the integer –1, which cannot be distinguished from EOF.

Use as array index: By similar reasoning, you cannot use sign-extended plain `char` variables as array index. If the sign bit is preserved, the conversion from `char` to `int` can result in negative integers. You must use positive integer values for array index.

Argument to character-handling function: By similar reasoning, you cannot use sign-extended plain `char` variables as arguments to character-handling functions declared in `ctype.h`, for instance, `isalpha()` or `isdigit()`. According to the C11 standard (Section 7.4), if you supply an integer argument that cannot be represented as `unsigned char` or EOF, the resulting behavior is undefined.

Polyspace Implementation

The check raises a flag when:

- You use invalid arguments with an integer function from the standard library. This check picks up:
  - Character Conversion

    ```
    toupper, tolower
    ```
  - Character Checks

    ```
    isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct,
    isspace, isupper, isxdigit
    ```
  - Integer Division

    ```
    div, ldiv
    ```
  - Absolute Values

    ```
    abs, labs
    ```
- You convert a signed or plain `char` data type to a wider integer data type with sign extension. You then use the resulting sign-extended value as array index, for comparison with EOF or as argument to a character-handling function.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Absolute Value of Large Negative

```
#include <limits.h>
#include <stdlib.h>

int absoluteValue(void) {

    int neg = INT_MIN;
    return abs(neg);
}
```

The input value to `abs` is `INT_MIN`. The absolute value of `INT_MIN` is `INT_MAX+1`. This number cannot be represented by the type `int`.

Correction — Change Input Argument

One possible correction is to change the input value to fit returned data type. In this example, change the input value to `INT_MIN+1`.

```
#include <limits.h>
#include <stdlib.h>

int absoluteValue(void) {

    int neg = INT_MIN+1;
    return abs(neg);
}
```

Sign-Extended Character Value Compared with EOF

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];

static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = *buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

In this example, the function `parser` can traverse a string input `buf`. If a character in the string has the value -1, it can represent either `EOF` or the valid character value `'\377'` (corresponding to the `unsigned char` equivalent 255). When converted to the `int` variable `c`, its value becomes the integer value -1, which is always `EOF`. The later comparison with `EOF` will not detect if the value returned from `parser` is actually `EOF`.

Correction — Cast to `unsigned char` Before Conversion

One possible correction is to cast the plain `char` value to `unsigned char` before conversion to the wider `int` type. Only then can you test if the return value of `parser` is really `EOF`.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];

static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = (unsigned char)*buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

Check Information

Group: Strings library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A23-0-1

An iterator shall not be implicitly converted to const_iterator.

Description

Rule Definition

An iterator shall not be implicitly converted to const_iterator.

Rationale

The C++11 standard introduces member functions such as `cbegin` and `cend` that returns const iterators to containers. To create const iterators, use these member functions instead of functions such as `begin` and `end` that return non-const iterators and then require implicit conversions.

For instance, consider the `std::list` container:

```
std::list<int> aList = {0, 0, 1, 2};
```

You can use the `begin` and `end` member functions of the container to create const iterators, for instance in a `for` loop:

```
for(std::vector<int>::const_iterator iter{aList.begin()}, end{aList.end()};
    iter != end;
    ++iter) {...}
```

However, the functions `begin` and `end` return non-const iterators and for assignment to the const iterators `iter` and `end` respectively, an implicit conversion must happen. Instead, take advantage of the new C++11 functions `cbegin` and `cend` that directly returns const iterators:

```
for(std::vector<int>::const_iterator iter{aList.cbegin()}, end{aList.cend()};
    iter != end;
    ++iter) {...}
```

If you use these functions, you can also replace the explicit type specification of the iterators with `auto`:

```
for(auto iter{aList.cbegin()}, end{aList.cend()};
    iter != end;
    ++iter) {...}
```

Polyspace Implementation

The checker flags conversions from type `iterator` to `const_iterator` or `reverse_iterator` to `const_reverse_iterator`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Implicit Conversions to Const Iterators

```
#include <cstdint>
#include <vector>

void func(std::vector<int32_t> & values, int32_t aValue) {
    std::vector<int32_t>::const_iterator iter1 =
                  std::find(values.begin(), values.end(), aValue); //Noncompliant
    std::vector<int32_t>::const_iterator iter2 =
                  std::find(values.cbegin(), values.cend(), aValue);  //Compliant
}
```

In this example, the first `std::find` function call uses as arguments the return values of the `begin` and `end` methods of an `std::vector` container `values`. These methods return iterators of type `std::vector<intr32_t>::iterator`. Since the `std::find` template has the same return type as the types of the first two arguments, it also returns an iterator of type `std::vector<intr32_t>::iterator`. The return value is assigned to a variable of type `std::vector<intr32_t>::const_iterator`, resulting in an implicit conversion.

The second call uses the `cbegin` and `cend` methods which return iterators of type `std::vector<intr32_t>::const_iterator` and avoid the implicit conversion.

Check Information

Group: Language support library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A2-3-1

Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code.

Description

Rule Definition

Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code.

Rationale

In the C++ standard, the basic source character set consists of 96 characters. They are:

- The space character.
- The control characters such as horizontal tab, vertical tab, form feed, and new line.
- Upper and lower case letters, and numbers.
- Special characters, such as `_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '`.

Using characters outside this set can cause confusion and unexpected bugs. For example, the Greek letter "T" is visually similar to the English letter "T", but they are separate characters with different unicode code-point values. To avoid unexpected behavior, use only the above specified characters in your source code, including comments and string literals. You can use characters outside this set in only two cases. You can use:

- Other characters inside the text of a wide string or a UTF-8 encoded string.
- The character `@` inside comments, the text of a wide string, or a UTF-8 encoded string.

Polyspace Implementation

Polyspace flags the characters in your source code that are not in the set of 96 characters specified in C++ standard, with two exceptions. Polyspace does not flag:

- Other characters inside the text of a wide string or a UTF-8 encoded string.
- The character `@` inside comments, the text of a wide string, or a UTF-8 encoded string.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Do Not Use Characters Outside the Specified Set

The following example demonstrates the Polyspace implementation of AUTOSAR rule A2-3-1.

```
#include <cstdint>
```

```
// @ brief foo function    //Compliant by exception
/* @ brief foo function */ //Compliant by exception

#if 0
@ This one is not in a comment  //Noncompliant
#endif
/*Define £ and € as currency */   // Noncompliant
#define CUR1 "£"   //Noncompliant
#define CUR2 "€"   //Noncompliant
void myfunction(char *str );
int  Total = 0;    //Complaint
int  Total = 0;    //Noncompliant
void foo()
{
    char *s1 = "Greek T  - normal string"; //Noncompliant
    wchar_t *s2 { L"Greek T @ wide string"};  //Compliant
    char *s3 = u8"Greek T @ UTF-8";          // Compliant
    char16_t *s4 = u"Greek T UTF-16";  //Noncomplaint
    char32_t *s5 = U"Greek T UTF-32";  //Noncomplaint
    char *s6 = "mail@company.com"; //Noncomplaint
    myfunction("Greek T");//Noncomplaint
    myfunction(s3);//Complaint
}

main(){
    // ..
}
```

If your code has characters that are not in the specified character set, Polyspace flags them. Note the global variables `Total` and `Total`. Even though it looks as if they are the same variable, they are two different variables because the latter starts with the Greek letter "T". Confusion between these two characters can lead to unexpected behavior. Because the Greek letter "T" is outside the standard set of characters, Polyspace flags every use of the character, even those in comments and string literals.

Polyspace flags every use of characters outside the specified set, with the following exceptions. You can use:

- Other characters inside a wide string such as `s2` or UTF-8 encoded string such as `s3`.
- The character `@` inside a wide string such as `s2`, a UTF-8 encoded string such as `s3`, or a comment.

Check Information

Group: Lexical conventions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A2-5-1

Trigraphs shall not be used.

Description

Rule Definition

Trigraphs shall not be used.

Rationale

You denote trigraphs with two question marks followed by a specific third character (for instance,`'??-'` represents a `'~'` (tilde) character and `'??)'` represents a `']'`). These trigraphs can cause accidental confusion with other uses of two question marks.

For instance, the string

```
"(Date should be in the form ??-??-??)"
```

is transformed to

```
"(Date should be in the form ~~]"
```

but this transformation might not be intended.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Lexical Conventions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A2-5-2

Digraphs shall not be used.

Description

Rule Definition

Digraphs shall not be used.

Rationale

Digraphs are a sequence of two characters that are supposed to be treated as a single character. The checker flags use of these digraphs:

- `<%`, indicating `[`
- `%>`, indicating `]`
- `<:`, indicating `{`
- `:>`, indicating `}`
- `%:`, indicating `#`
- `%:%:`

When developing or reviewing code with digraphs, the developer or reviewer can incorrectly consider the digraph as a sequence of separate characters.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Lexical Conventions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A2-10-6

A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope.

Description

Rule Definition

A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope.

Rationale

When a variable, data member, function, or enumerator shares its name with a class or enumeration in the same scope, the latter is hidden. That is, all uses of the name refers to the variable, data member, function, or enumerator instead of the class or enumeration, regardless of declaration order. Hidden classes or enumerations can be misleading and can lead to compilation errors. Do not re-use names to declare classes and enumerations.

Polyspace Implementation

Polyspace flags the declaration of a variable, data member, function, or enumerator that shares the name of a class or enumeration in the same block.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Do Not Hide Class Declarations

The following example demonstrate the Polyspace implementation of AUTOSAR rule A2-10-6.

```
#include <cstdint>
namespace NS1
{
    class G {};
    void G() {}             //Noncompliant
}
namespace NS2
{
    enum class H { VALUE=0, };
    std::uint8_t H = 17;    //Noncompliant
}
namespace NS3
{
    class J {};
    enum H {
        J=0,                 // Noncompliant
    };
}
```

```
main()
{
    //...
}
```

Polyspaceflags the declaration of the:

- Function `G()` because it hides the class `G` declared in the same block.
- Variable `H` because it hides the enumeration `H` declared in the same block.
- Enumerator `J` because it hides the class `J` is declared in the same block.

Check Information

Group: Lexical conventions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A26-5-1

Pseudorandom numbers shall not be generated using std::rand().

Description

Rule Definition

Pseudorandom numbers shall not be generated using std::rand().

Rationale

This cryptographically weak routines is predictable and must not be used for security purposes. When a predictable random value controls the execution flow, your program is vulnerable to malicious attacks.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Random Loop Numbers

```
#include <stdio.h>
#include <stdlib.h>

volatile int rd = 1;
int main(int argc, char *argv[])
{
    int j, r, nloops;
    struct random_data buf;
    int i = 0;

    nloops = rand();

    for (j = 0; j < nloops; j++) {
        if (random_r(&buf, &i))
            exit(1);
        printf("random_r: %ld\n", (long)i);
    }
    return 0;
}
```

This example uses `rand` and `random_r` to generate random numbers. If you use these functions for security purposes, these PRNGs can be the source of malicious attacks.

Correction — Use Stronger PRNG

One possible correction is to replace the vulnerable PRNG with a stronger random number generator.

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <openssl/rand.h>

volatile int rd = 1;
int main(int argc, char* argv[])
{
    int j, r, nloops;
    unsigned char buf;
    unsigned int seed;
    int i = 0;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <seed> <nloops>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    seed = atoi(argv[1]);
    nloops = atoi(argv[2]);

    for (j = 0; j < nloops; j++) {
        if (RAND_bytes(&buf, i) != 1)
            exit(1);
        printf("RAND_bytes: %u\n", (unsigned)buf);
    }
    return 0;
}
```

Check Information

Group: Algorithms library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A2-7-1

The character \ shall not occur as a last character of a C++ comment.

Description

Rule Definition

The character \ shall not occur as a last character of a C++ comment.

Rationale

If your code has the character \ at the end of a single-line comment, the next line of code becomes a continuation of the comment. Ending single line comments by using the character \ can inadvertently comment-out sections of code.

Polyspace Implementation

Polyspace checks if the character \ is the last character of a C++ comment .

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Avoid Using the character \ as Last Character of C++ Comments

```
#include <cstdint>

void foo()
{
  int32_t idx = 0;
  int32_t limit = 20;
  int32_t count = 20;
  ++idx; // Incrementing index before the loop starts// Requirement X\\
  for(;idx<limit;++idx)
  {
     --count;
  }
}
```

The `for` loop definition is commented-out because the single-line comment ends with the character \. As a result, `count` is decremented only once, perhaps inadvertently. The checker flags this issue by highlighting the character \ in the single-line comment.

Check Information

Group: Lexical conventions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A2-13-6

Universal character names shall be used only inside character or string literals.

Description

Rule Definition

Universal character names shall be used only inside character or string literals.

Rationale

Universal character names are a way to represent unicode characters by using code points. For example, `\U0000231A` represents the unicode character '⌚'. When you use universal character names to define an identifier, it is difficult to read the source code . Some environments allow the use of universal character names and their literal forms interchangeably, which makes the code confusing and can lead to subsequent errors. Avoid using universal character names outside a character or string literal.

Polyspace Implementation

Polyspace flags the use of universal character names outside a character or string literal.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Do Not Use Universal Character Names Outside Character or String Literal

The following example demonstrate the Polyspace implementation of AUTOSAR rule A2-13-6.

```
#include <cstdint>
#define \U0000231AMACRO(x) (x)     // Noncompliant
int \U000030AD = 0;   //Noncompliant
void €uro(){            // Compliant
    std::int32_t €uro;    // Compliant
    std::int32_t \U0000231Ahello; // Noncompliant
    wchar_t wc = '\U0000231A';    // Compliant
    std::int32_t Hello\U0000231AWorld;  // Noncompliant
       キ=2; // \u30AD is 'キ'
}
typedef struct \U0000231Astruct {    // Noncompliant
    std::int32_t regular;
    std::int32_t €uro;                // Compliant
    std::int32_t \U0000231Ahello;     // Noncompliant
} \U0001f615type;                // Noncompliant

main(){
    //...
}
```

The variable `\U000030AD` is initialized to 0. the, it is later referred to by using its literal form 'キ'. It is unclear that キ and `\U000030AD` refer to the same global variable. To avoid confusion caused by universal character names, Polyspace flags their use outside a character or string literal.

Check Information

Group: Lexical conventions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A3-1-1

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

Description

Rule Definition

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

Rationale

If a header file with variable or function definitions appears in multiple inclusion paths, the header file violates the One Definition Rule possibly leading to unpredictable behavior. For instance, a source file includes the header file `include.h` and another header file, which also includes `include.h`.

Polyspace Implementation

The rule checker flags variable and function definitions in header files.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Basic Concepts

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A3-1-2

Header files, that are defined locally in the project, shall have a file name extension of one of: `.h`, `.hpp` or `.hxx`

Description

Rule Definition

Header files, that are defined locally in the project, shall have a file name extension of one of: `.h`, `.hpp` or `.hxx`.

Rationale

Developers and code reviewers expect a header file to have one of the standard file name extensions.

Polyspace Implementation

The rule checker flags files included with the `#include` directive with names that have an extension other than `.h`, `.hpp` or `.hxx`. For instance:

```
#include <header.c>
#include <header2.cpp>
```

Instead of `<...>`, if you use `"..."` around the file, the checker also flags the case where the file does not have an extension at all.

The checker does not flag the following inclusions:

- Files included with the `Include (-include)` option.
- Included files that do not exist.

The checker is case-insensitive.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Basic Concepts

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019b

AUTOSAR C++14 Rule A3-1-3

Implementation files, that are defined locally in the project, should have a file name extension of ".cpp".

Description

Rule Definition

Implementation files, that are defined locally in the project, should have a file name extension of ".cpp*".*

Polyspace Implementation

Not case sensitive if you set the option `-dos`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A3-1-4

When an array with external linkage is declared, its size shall be stated explicitly.

Description

Rule Definition

When an array with external linkage is declared, its size shall be stated explicitly.

Rationale

Though you can declare an incomplete array type and later complete the type, specifying the array size during the first declaration makes the subsequent array access less error-prone.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Array Size Unspecified During Declaration

```
int array[10];
extern int array2[]; //Noncompliant
int array3[]= {0,1,2};
extern int array4[10];
```

In the declaration of `array2`, the array size is unspecified.

Check Information

Group: Basic Concepts

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A3-3-1

Objects or functions with external linkage (including members of named namespaces) shall be declared in a header file.

Description

Rule Definition

Objects or functions with external linkage (including members of named namespaces) shall be declared in a header file.

Rationale

If you declare a function or object in a header file, it is clear that the function or object is meant to be accessed in multiple translation units. If you intend to access the function or object from a single translation unit, declare it `static` or in an unnamed namespace.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Declaration in Header File Missing

This example uses two files:

- `decls.h`:

  ```
  extern int x;
  ```

- `file.cpp`:

  ```
  #include "decls.h"

  int x = 0;
  int y = 0; //Noncompliant
  static int z = 0;
  ```

In this example, the variable `x` is declared in a header file but the variable `y` is not. The variable `z` is also not declared in a header file but it is declared with the `static` specifier and does not have external linkage.

Check Information

Group: Basic Concepts

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A3-3-2

Static and thread-local objects shall be constant-initialized.

Description

Rule Definition

Static and thread-local objects shall be constant-initialized.

Rationale

Static and thread-local objects are initialized at the start of code execution. The C++ language standard only partially defines the initialization order of multiple static or thread-local objects and the order can change from build to build. If you initialize a static or thread-local object from another such object, the compiler might access the latter object before it is initialized. To avoid access before initialization, initialize static and thread-local objects by using objects that evaluate to a constant at compile time. Initialization with constants occurs before initialization with variables and often happens at compile time.

This rule applies to global variables, static variables, static class member variables, and static function-scope variables.

Polyspace Implementation

Polyspace flags initializations of static or thread-local objects using initializers and constructors that do not evaluate to constants at compile time. To constant-initialize static or thread-local objects, use:

- A `constexpr` constructor with only constant arguments
- A constant expression
- A value

Because string objects use dynamic memory allocation of unknown size, the compiler cannot evaluate them at compile time. Polyspace flags initialization of string objects irrespective of whether you specify an initializer.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Initializing Static and Thread-Local Objects

```
#include <cstdint>
#include <limits>
#include <string>

const int global_const_a = 10;              // Compliant
const int global_const_b = global_const_a;  // Compliant
int global_a = 10;                          // Compliant
int global_b = global_a;                    // Noncompliant
```

```
static std::string global_name = "Name";    // Noncompliant
static std::string global_id;               // Noncompliant
char *ptr = "hello world";                  // Compliant
char arr_up[3] = {'U','p','\0'};            // Compliant
char container[10];                         // Compliant
extern const int global_extern_c;
const int global_const_c = global_extern_c; // Noncompliant
  main()
  {

    //
  }
```

Polyspace flags the initialization of:

- `global_b` by `global_a` because whether `global_b` evaluates to a constant at compile time depends on the order in which these variables are initialized.
- `global_name` and `global_id` because the compiler cannot evaluate constructor for string objects at compile time.
- `global_const_c` by the `extern` variable `global extern_c` because the compiler cannot evaluate `extern` variables at compile time.

Polyspace does not flag the initialization of:

- `global_const_b` by `global_const_a` because the compiler can evaluate these objects at compile time regardless of their initialization order.
- `global_const_a` and `global_a` by literals because the compiler can evaluate literals at compile time.
- Global character pointers and arrays by literal initializers because the compiler can allocate static memory at compile time.

Check Information

Group: Basic concepts

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A3-9-1

Fixed width integer types from <cstdint>, indicating the size and signedness, shall be used in place of the basic numerical types.

Description

Rule Definition

Fixed width integer types from <cstdint>, indicating the size and signedness, shall be used in place of the basic numerical types.

Polyspace Implementation

Only allows use of basic types through direct `typedefs`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A4-10-1

Only nullptr literal shall be used as the null-pointer-constraint.

Description

Rule Definition

Only nullptr literal shall be used as the null-pointer-constraint.

Rationale

`nullptr` was introduced in C++11 to support the concept of a pointer that does not point to a valid object. Before C++11, the macro `NULL` and the constant 0 were the only ways to define the null pointer constant. However, these alternatives to `nullptr` can also be used in integer contexts and can cause potential confusion.

For instance, if a function is overloaded with an integer and pointer argument:

```
void foo(int);
void foo(int*);
```

using the argument `NULL` as:

```
foo(NULL);
```

can cause confusion about which function is called. In this example, the integer-argument overload is called but a developer or reviewer might expect otherwise.

Polyspace Implementation

The rule checker flags uses of the `NULL` macro or the constant 0 as pointers (via direct assignment, casts or otherwise).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `NULL` or 0 as Alternatives to `nullptr`

```
#include <cstdint>
#include <cstddef>


void foo(int*);
void foo2(int*);

void bar() {
    foo(NULL);    //Noncompliant
    foo2(0);      //Noncompliant
    foo(nullptr); //Compliant
```

```
}
```

In this example, the rule is violated when the macro `NULL` or the constant 0 is used as a pointer.

Check Information

Group: Standard conversions

See Also

`AUTOSAR C++14 Rule M4-10-2` | `AUTOSAR C++14 Rule M4-10-1` | `Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A4-5-1

Expressions with type enum or enum class shall not be used as operands to built-in and overloaded operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.

Description

Rule Definition

Expressions with type enum or enum class shall not be used as operands to built-in and overloaded operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.

Rationale

In C++, enumerations such as `enum` or `enum class` have implementations defined behavior. For instance, their underlying type can be any integral type, including `short` or `char`. If you use enumerations as operands to arithmetic operators such as `+` or `-`, they are converted to their underlying type. Because the underlying type of an enumeration is implementation dependent, outcome of arithmetic operations using enumerations as operands is unpredictable. To avoid unpredictable and non-portable code, use enumerations as operands to only these operators:

- Subscript operator `[ ]`
- Assignment operator `=`
- Equality operators `==` and `!=`
- The Unary `&` operator
- The relational operators `<, <=,>,>=`

You can use enumerations as operands to the built in or overloaded instances of only the above operators. Note that Bitmask type enumerations are an exception to this rule. That is, you can use Bitmask type enumerations as operands to any operators.

Polyspace Implementation

Enumerations are valid operands to only the operators listed above. Polyspace flags enumerations when they are used as operands to any other operators. Note that Polyspace makes no exception for BitmaskType enumerations.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Do Not Use Enumerations as Operands to Arithmetic Operators

```
#include <cstdint>
enum Color : std::uint8_t {  Red,  Green,  Blue,  ColorsCount};
enum class Car : std::uint8_t { Model1, Model2, Model3, ModelsCount};
enum BMT {Exec = 0x1,Write  = 0x2,Read = 0x4};
```

```
Car operator+(Car lhs, Car rhs)
{
    return Car::Model3;
}
Color operator|=(Color lhs, Color rhs)
{
    return rhs;
};
void F1()
{
    Car car = Car::Model1;
    Color color = Red;
    if (color == Green) {                   // Compliant
    }

    if (color == (Red + Blue)) {            // Noncompliant
    }

    if (color < ColorsCount) {              //Compliant
    }
    if (car == (Car::Model1 + Car::Model2)) // Noncompliant
    {
    }
    Color value;
    value = (Color)(Red | 3);               // Noncompliant
    value |= Blue;                          // Noncompliant
    value = (Color)0;                       // Noncompliant
    if (value & Blue) {};                   // Noncompliant
    value = (Color)(Blue * value       );   // Noncompliant
    value = (Color)(Red << 3);              // Noncompliant
    value = (Color)(Red >> 12);             // Noncompliant
    BMT bitmask1 = (BMT)(Exec + Write);     // Noncompliant
    BMT bitmask2 = (BMT)(Exec | Write);     // Noncompliant
}
```

The line `BMT bitmask1 = (BMT)(Exec + Write);` adds two enumerators and assigns the result to the `enum` object `bitmask1`. The addition operation implicitly converts the enumerators into their underlying type. Because the underlying type of enumerators are implementation dependent, the outcome of this code can be unpredictable. Polyspace flags the enumerators that are operands to the built in + operator.

Polyspace treats both built in and overloaded operators similarly. For example, Polyspace flags the operands in the operation `Car::Model1 + Car::Model2`, even though the + operator is overloaded for the `enum class Car`.

Check Information

Group: Standard conversions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A5-0-1

The value of an expression shall be the same under any order of evaluation that the standard permits.

Description

Rule Definition

The value of an expression shall be the same under any order of evaluation that the standard permits.

Rationale

If an expression results in different values depending on the order of evaluation, its value becomes implementation-defined.

Polyspace Implementation

An expression can have different values under the following conditions:

- The same variable is modified more than once in the expression, or is both read and written.
- The expression allows more than one order of evaluation.

Therefore, the rule checker forbids expressions where a variable is modified more than once and can cause different results under different orders of evaluation. The rule checker also detects cases where a volatile variable is read more than once in an expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Variable Modified More Than Once in Expression

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);         /* Compliant */
    COPY_ELEMENT (i++);       /* Non-compliant  */
}
```

In this example, the rule is violated by the statement `COPY_ELEMENT(i++)` because `i++` occurs twice and the order of evaluation of the two expressions is unspecified.

Variable Modified and Used in Multiple Function Arguments

```
void f (unsigned int param1, unsigned int param2) {}

void main () {
    unsigned int i=0;
```

```
    f ( i++, i );                 /* Non-compliant */
}
```

In this example, the rule is violated because it is unspecified whether the operation `i++` occurs before or after the second argument is passed to `f`. The call `f(i++,i)` can translate to either `f(0,0)` or `f(0,1)`.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A5-0-2

The condition of an if-statement and the condition of an iteration statement shall have type bool.

Description

Rule Definition

The condition of an if-statement and the condition of an iteration statement shall have type bool.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A5-0-3

The declaration of objects shall contain no more than two levels of pointer indirection.

Description

Rule Definition

The declaration of objects shall contain no more than two levels of pointer indirection.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A5-0-4

Pointer arithmetic shall not be used with pointers to non-final classes.

Description

Rule Definition

Pointer arithmetic shall not be used with pointers to non-final classes.

Polyspace Implementation

Reports pointer arithmetic and array like access on expressions whose pointed type is used as a base class.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A5-1-1

Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead.

Description

Rule Definition

Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead.

Rationale

It is often unclear from use of literal constants what the constant represents. Using named constants improves the readability of the code.

Polyspace Implementation

The rule checker flags use of literal constants other than those with data type `char` in expressions, non-`const` initializations and `case` clauses of a `switch` statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A5-1-2

Variables shall not be implicitly captured in a lambda expression.

Description

Rule Definition

Variables shall not be implicitly captured in a lambda expression.

Rationale

In a lambda expression, you have the option to capture variables implicitly. For instance, this lambda expression

```
[&](std::int32_t var) {
   sum+ = var;
}
```

indicates that all local variables in the calling context are captured by reference. However, it is not immediately clear from this lambda expression:

- If a variable in the body of the expression comes from the calling context.

  For instance, in the preceding lambda expression, it is not clear if `sum` is captured from the calling context or is a global variable.
- If all variables captured from the calling context are used and whether the variables are modified or just read (If the variables are read, a by-copy capture is preferred).

If you capture variables explicitly in a lambda expression, you have more control on whether to capture by reference or copy. In addition, you or a reviewer can read the lambda expression and determine whether a variable was captured from the calling context.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Lambda Expressions with Implicit and Explicit Capture

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <cstdint>

void addEvenNumbers(std::vector<std::int32_t> numbers)
{
    std::int64_t sum = 0;
    std::int32_t divisor = 2;
    for_each(numbers.begin(), numbers.end(), [&] (std::int32_t y) //Noncompliant
    {
        if (y % divisor == 0)
```

```
        {
            std::cout << y << std::endl;
            sum += y;
        }
    });

    std::cout << sum << std::endl;
}

void addOddNumbers(std::vector<std::int32_t> numbers)
{
    std::int64_t sum = 0;
    std::int32_t divisor = 2;
    for_each(numbers.begin(), numbers.end(), [&sum, divisor] (std::int32_t y) //Compliant
    {
        if (y % divisor != 0)
        {
            std::cout << y << std::endl;
            sum += y;
        }
    });

    std::cout << sum << std::endl;
}
```

The lambda expression in the `addEvenNumbers` function captures all local variables in the calling context implicitly by reference and violates this rule. Some of the issues are:

- Unless you go through the body of the expression, it is not clear which variables are used.
- Though the variable `divisor` is only read and not modified, it is captured by reference. A by-copy capture is preferred.

The lambda expression in the `addOddNumbers` function captures each variable explicitly and does not violate this rule. Without looking at the body of the lambda expression, you can determine which variables are intended to be modified in the expression.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019b

AUTOSAR C++14 Rule A5-1-3

Parameter list (possibly empty) shall be included in every lambda expression.

Description

Rule Definition

Parameter list (possibly empty) shall be included in every lambda expression.

Rationale

You do not have to include a parameter list in a lambda expression. For instance, this expression is syntactically valid and indicates a closure that can be called without parameters:

```
[&counter] {
   ++counter;
}
```

However, without the `()`, you or a reviewer might not recognize this as a function object. It is visually clearer to use the parameter list `(...)` even when the list is empty. For instance:

```
[&counter]() {
   ++counter;
}
```

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Lambda Expressions Without Parameter List

```
#include <cstdint>

void func() {
    std::int32_t count = 0;

    auto lambda1 = [&count] {++count;}; //Noncompliant
    auto lambda2 = [&count] () { //Compliant
        ++count;
    };
}
```

The lambda expression assigned to `lambda1` does not have a parameter list and violates the rule. The issue is fixed when the same lambda expression is assigned to `lambda2`.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019b

AUTOSAR C++14 Rule A5-1-4

A lambda expression object shall not outlive any of its reference-captured objects.

Description

Rule Definition

A lambda expression object shall not outlive any of its reference-captured objects.

Rationale

The rule flags cases where a lambda expression captures an object *by reference* and you can potentially access the captured object outside its scope. This situation happens if the lambda expression object outlives the object captured by reference.

For instance, consider this function `createFunction`:

```
std::function<std::int32_t()> createFunction() {
   std::int32_t localVar = 0;
   return ([&localVar]() -> std::int32_t {
       localVar = 1;
       return localVar;
   });
}
```

`createFunction` returns a lambda expression object that captures the local variable `localVar` *by reference*. The scope of `localVar` is limited to `createFunction` but the lambda expression object returned has a much larger scope.

This situation can result in an attempt to access the local object `localVar` outside its scope. For instance, when you call `createFunction` and assign the returned lambda expression object to another object `aFunction`:

```
auto aFunction = createFunction();
```

and then invoke the new object `aFunction`:

```
std::int32_t someValue = aFunction();
```

the captured variable `localVar` is no longer in scope. Therefore, the value returned from `aFunction` is undefined.

If a function returns a lambda expression, to avoid accessing a captured object outside its scope, make sure that the lambda expression captures all objects by copy. For instance, you can rewrite `createFunction` as:

```
std::function<std::int32_t()> createFunction() {
   std::int32_t localVar = 0;
   return ([localVar]() mutable -> std::int32_t {
       localVar = 1;
       return localVar;
   });
}
```

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019b

AUTOSAR C++14 Rule A5-1-7

A lambda shall not be an operand to decltype or typeid.

Description

Rule Definition

A lambda shall not be an operand to decltype or typeid.

Rationale

According to the C++ Standard, the type of a lambda expression is a unique, unnamed class type. Because the type is unique, another variable or expression cannot have the same type. Use of `decltype` or `typeid` on a lambda expression indicates that you expect a second variable or expression to have the same type as the operand lambda expression.

Both `decltype` and `typeid` return the data type of their operands. Typically the operators are used to:

- Assign a type to another variable. For instance:

  ```
  decltype(var1) var2;
  ```

  creates a variable `var2` with the same type as `var1`.
- Compare the types of two variables. For instance:

  ```
  (typeid(var1) == typeid(var2))
  ```

  compares the types of `var1` and `var2`.

These uses do not apply to a lambda expression, which has a unique type.

Polyspace Implementation

The rule checker flags uses of `decltype` and `typeid` with lambda expressions.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `typeid` on Lambda Expressions

```
#include <cstdint>
#include <typeinfo>

 void func()
 {
 auto lambdaFirst = []() -> std::int8_t { return 1; };
 auto lambdaSecond = []() -> std::int8_t { return 1; };
```

```
 if (typeid(lambdaFirst) == typeid(lambdaSecond))
     {
     // ...
     }
 }
```

The use of `typeid` on lambda expressions can lead to unexpected results. The comparison above is false even though `lambdaFirst` and `lambdaSecond` appear to have the same body.

Correction – Assign Lambda Expression to Function Object Before Using `typeid`

One possible correction is to assign the lambda expression to a function object and then use the `typeid` operator on the function objects for comparison.

```
#include <cstdint>
#include <functional>
#include <typeinfo>

 void func()
 {
 std::function<std::int8_t()> functionFirst = []() { return 1; };
 std::function<std::int8_t()> functionSecond = []() { return 1; };

 if (typeid(functionFirst) == typeid(functionSecond))
     {
     // ...
     }
 }
```

Check Information

Group: Lexical Conventions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019b

AUTOSAR C++14 Rule A5-2-2

Traditional C-style casts shall not be used.

Description

Rule Definition

Traditional C-style casts shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A5-2-3

A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

Description

Rule Definition

A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A5-2-4

reinterpret_cast shall not be used

Description

Rule Definition

reinterpret_cast shall not be used.

Rationale

`reinterpret_cast` is typically used to explicitly convert between two unrelated data types. For instance, in this example, `reinterpret_cast` converts the type `struct S*` to `int*`:

```
struct S { int x; } s;
int* ptr = reinterpret_cast<int*> (&s);
```

However, it is difficult to use `reinterpret_cast` and not violate type safety. If the result of `reinterpret_cast` is a pointer, it is safe to dereference the pointer only after you cast the pointer back to its original type.

Polyspace Implementation

The rule checker flags all uses of the `reinterpret_cast` keyword.

If the rule checker flags an use of `reinterpret_cast` that you consider safe, add a comment justifying the result. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `reinterpret_cast` Keyword

```
class A {
    int x;
    int y;
    public:
      void getxy();
};

class B {
    int z;
    public:
      void getz();
};

void func (B* Bptr) {
    A* Aptr = reinterpret_cast<A*>(Bptr);
}
```

The use of `reinterpret_cast` violates this rule. The result of `reinterpret_cast` is not safe to dereference since `A` and `B` are unrelated classes. Dereferencing `Aptr` as if it were an `A*` pointer can result in illegal memory access.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A5-2-6

The operands of a logical && or || shall be parenthesized if the operands contain binary operators.

Description

Rule Definition

The operands of a logical && or \|\| shall be parenthesized if the operands contain binary operators.

Polyspace Implementation

During preprocessing, violations of this rule are detected on the expressions in `#if` directives.

The checker allows exceptions on associativity (a && b && c), (a || b || c).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A5-3-3

Pointers to incomplete class types shall not be deleted.

Description

Rule Definition

Pointers to incomplete class types shall not be deleted.

Rationale

When you delete a pointer to an incomplete class, it is not possible to call any nontrivial destructor that the class might have. If the destructor performs cleanup activities such as memory deallocation, these activities do not happen.

A similar problem happens, for instance, when you downcast to a pointer to an incomplete class (downcasting is casting from a pointer to a base class to a pointer to a derived class). At the point of downcasting, the relationship between the base and derived class is not known. In particular, if the derived class inherits from multiple classes, at the point of downcasting, this information is not available. The downcasting cannot make the necessary adjustments for multiple inheritance and the resulting pointer cannot be dereferenced.

Polyspace Implementation

The check raises a defect when you delete or cast to a pointer to an incomplete class. An incomplete class is one whose definition is not visible at the point where the class is used.

For instance, the definition of class `Body` is not visible when the `delete` operator is called on a pointer to `Body`:

```
class Handle {
  class Body *impl;
public:
  ~Handle() { delete impl; }
  // ...
};
```

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Deletion of Pointer to Incomplete Class

```
class Handle {
  class Body *impl;
public:
  ~Handle() { delete impl; }
  // ...
};
```

In this example, the definition of class `Body` is not visible when the pointer to `Body` is deleted.

Correction — Define Class Before Deletion

One possible correction is to make sure that the class definition is visible when a pointer to the class is deleted.

```
class Handle {
  class Body *impl;
public:
  ~Handle();
  // ...
};

// Elsewhere
class Body { /* ... */ };

Handle::~Handle() {
  delete impl;
}
```

Correction — Use `std::shared_ptr`

Another possible correction is to use the `std::shared_ptr` type instead of a regular pointer.

```
#include <memory>

class Handle {
  std::shared_ptr<class Body> impl;
  public:
    Handle();
    ~Handle() {}
    // ...
};
```

Downcasting to Pointer to Incomplete Class

`File1.h`:

```
class Base {
protected:
  double var;
public:
  Base() : var(1.0) {}
  virtual void do_something();
  virtual ~Base();
};
```

`File2.h`:

```
void funcprint(class Derived *);
class Base *get_derived();
```

`File1.cpp`:

```
#include "File1.h"
#include "File2.h"

void getandprint() {
  Base *v = get_derived();
```

```
  funcprint(reinterpret_cast<class Derived *>(v));
}
```

`File2.cpp`:

```
#include "File2.h"
#include "File1.h"
#include <iostream>

class Base2 {
protected:
  short var2;
public:
  Base2() : var2(12) {}
};

class Derived : public Base2, public Base {
  float var_derived;
public:
    Derived() : Base2(), Base(), var_derived(1.2f) {}
    void do_something()
    {
        std::cout << "var_derived: "
                  << var_derived << ", var : " << var
                  << ", var2: " << var2 << std::endl;
    }
 };

void funcprint(Derived *d) {
  d->do_something();
}

Base *get_derived() {
  return new Derived;
}
```

In this example, the definition of class `Derived` is not visible in `File1.cpp` when a `Base*` pointer to downcast to a `Derived*` pointer.

In `File2.cpp`, class `Derived` derives from two classes, `Base` and `Base2`. This information about multiple inheritance is not available at the point of downcasting in `File1.cpp`. The result of downcasting is passed to the function `funcprint` and dereferenced in the body of `funcprint`. Because the downcasting was done with incomplete information, the dereference can be invalid.

Correction — Define Class Before Downcasting

One possible correction is to define the class `Derived` before downcasting a `Base*` pointer to a `Derived*` pointer.

In this corrected example, the downcasting is done in `File2.cpp` in the body of `funcprint` at a point where the definition of class `Derived` is visible. The downcasting is not done in `File1.cpp` where the definition of `Derived` is not visible. The changes from the previous incorrect example are highlighted.

`File1.h`:

```
class Base {
protected:
```

```
  double var;
public:
  Base() : var(1.0) {}
  virtual void do_something();
  virtual ~Base();
};
```

File2.h:

```
void funcprint(class Base *);
class Base *get_derived();
```

File1.cpp:

```
#include "File1.h"
#include "File2.h"

void getandprint() {
  Base *v = get_derived();
  funcprint(v);
}
```

File2.cpp:

```
#include "File2_corr.h"
#include "File1_corr.h"
#include <iostream>

class Base2 {
protected:
  short var2;
public:
  Base2() : var2(12) {}
};

class Derived : public Base2, public Base {
  float var_derived;

public:
    Derived() : Base2(), Base(), var_derived(1.2f) {}
    void do_something()
    {
        std::cout << "var_derived: "
                  << var_derived << ", var : " << var
                  << ", var2: " << var2 << std::endl;
    }
};

void funcprint(Base *d) {
  Derived *temp = dynamic_cast<Derived*>(d);
  if(temp)  {
     d->do_something();
  }
  else {
      //Handle error
  }
}
```

```
Base *get_derived() {
  return new Derived;
}
```

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A5-6-1

The right hand operand of the integer division or remainder operators shall not be equal to zero.

Description

Rule Definition

The right hand operand of the integer division or remainder operators shall not be equal to zero.

Rationale

- If the numerator is the minimum possible value and the denominator is `-1`, your division operation overflows because the result cannot be represented by the current variable size.
- If the denominator is zero, your division operation fails possibly causing your program to crash.

These risks can be used to execute arbitrary code. This code is usually outside the scope of a program's implicit security policy.

- If the second remainder operand is zero, your remainder operation fails, causing your program to crash.
- If the second remainder operand is `-1`, your remainder operation can overflow if the remainder operation is implemented based on the division operation that can overflow.
- If one of the operands is negative, the operation result is uncertain. For C89, the modulo operation is not standardized, so the result from negative operands is implementation-defined.

These risks can be exploited by attackers to gain access to your program or the target in general.

Polyspace Implementation

The checker raises a defect when:

- The denominator of a division or modulo operation can be a zero-valued integer.
- There are division operations where one or both of the integer operands is from an unsecure source.
- There are modulo operations with one or more tainted operands.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Dividing an Integer by Zero

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    result = num/denom;
```

```
    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

Correction — Check Before Division

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    if (denom != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
int fraction(int num)
{
    int denom = 2;
    int result = 0;

    result = num/denom;

    return result;
}
```

Modulo Operation with Zero

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        arr[i] = input % i;
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

In this example, Polyspace flags the modulo operation as a division by zero. Because modulo is inherently a division operation, the divisor (right hand argument) cannot be zero. The modulo operation uses the `for` loop index as the divisor. However, the `for` loop starts at zero, which cannot be an iterator.

Correction — Check Divisor Before Operation

One possible correction is checking the divisor before the modulo operation. In this example, see if the index `i` is zero before the modulo operation.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        if(i != 0)
        {
             arr[i] = input % i;
        }
        else
        {
             arr[i] = input;
        }
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

Correction — Change Divisor

Another possible correction is changing the divisor to a nonzero integer. In this example, add one to the index before the % operation to avoid dividing by zero.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
         arr[i] = input % (i+1);
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

Division of Function Arguments

```
extern void print_int(int);

int taintedintdivision(int usernum, int userden) {
    int r =  usernum/userden;
    print_int(r);
    return r;
}
```

This example function divides two argument variables, then prints and returns the result. The argument values are unknown and can cause division by zero or integer overflow.

Correction — Check Values

One possible correction is to check the values of the numerator and denominator before performing the division.

```
#include "limits.h"

extern void print_int(int);

int taintedintdivision(int usernum, int userden) {
    int r = 0;
    if (userden!=0 && !(usernum=INT_MIN && userden==-1)) {
```

```
        r = usernum/userden;
    }
    print_int(r);
    return r;
}
```

Modulo with Function Arguments

```
extern void print_int(int);

int taintedintmod(int userden) {
    int rem =  128%userden;
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using an input argument. The argument is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

Correction — Check Operand Values

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
extern void print_int(int);

int taintedintmod(int userden) {
    int rem = 0;
    if (userden > 0) {
        rem = 128 % userden;
    }
    print_int(rem);
    return rem;
}
```

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A5-16-1

The ternary conditional operator shall not be used as a sub-expression.

Description

Rule Definition

The ternary conditional operator shall not be used as a sub-expression.

Rationale

A ternary conditional operator used as a subexpression makes the full expression less readable and difficult to maintain. It is often visually clearer if you assign the result of a ternary operator to a variable and then use the variable in subsequent operations.

Polyspace Implementation

The checker flags uses of the ternary conditional operator in subexpressions with some exceptions. Exceptions include uses of the operator when:

- The result is assigned to a variable.
- The result is used as a function argument or returned from a function.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of Ternary Operators as Sub-expressions

```
#include <cstdint>
const int ULIM = 100000;

std::int32_t foo(int32_t x) {
    int ret;
    ret = (x <= 0? 0: (x >= ULIM? 0 : x)); //Noncompliant
    return ret;
}

std::int32_t bar(int32_t x) {
    int ret, retInterim;
    retInterim = x >= ULIM? 0 : x; //Compliant
    ret = retInterim <= 0? 0 : retInterim; //Compliant
    return ret;
}
```

In this example, in `foo`, a ternary conditional operation is chained with a second operation to return the value 0 if `x` is in the range `[0, ULIM]` and return `x` otherwise. The ternary operation comparing `x` with `ULIM` is a sub-expression in the full chain and violates the rule.

In `bar`, each ternary conditional operation is written in a separate step and does not violate the rule. Alternatively, the same algorithm can be implemented by combining the conditions with the boolean AND operator and using a single ternary conditional operation.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A6-4-1

A switch statement shall have at least two case-clauses, distinct from the default label.

Description

Rule Definition

A switch statement shall have at least two case-clauses, distinct from the default label.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A6-5-2

A for loop shall contain a single loop-counter which shall not have floating-point type.

Description

Rule Definition

A for loop shall contain a single loop-counter which shall not have floating-point type.

Polyspace Implementation

The checker flags these situations:

- The `for` loop index has a floating point type.
- More than one loop counter is incremented in the `for` loop increment statement.

  For instance:

  ```
  for(i=0, j=0; i<10 && j < 10;i++, j++) {}
  ```
- The `for` loop increment statement is missing.

  For instance:

  ```
  for(i=0; i<10;) {}
  ```

  Even if you increment the loop counter in the loop body, the checker still raises a violation. The rule is based on MISRA C++ rule 6-5-1. According to the MISRA C++ specifications, a loop counter is one that is initialized in or prior to the loop expression, acts as an operand to a relational operator in the loop expression and *is modified in the loop expression*. If the increment statement in the loop expression is missing, the checker cannot find the loop counter modification and considers as if a loop counter is not present.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A6-5-4

For-init-statement and expression should not perform actions other than loop-counter initialization and modification.

Description

Rule Definition

For-init-statement and expression should not perform actions other than loop-counter initialization and modification.

Polyspace Implementation

- Reports if `loop` parameter cannot be determined. Assumes JSF C++ Rule 200 is not violated. The `loop variable` parameter is assumed to be a variable.
- Assumes 1 loop parameter (see JSF C++ Rule 198), with non class type. JSF C++ Rule 200 must not be violated for this rule to be reported.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A6-6-1

The goto statement shall not be used.

Description

Rule Definition

The goto statement shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A7-1-3

CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name

Description

Rule Definition

CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name.

Rationale

Suppose a `typedef` or `using` statement defines a pointer type. For instance:

```
using IntPtr = std::int32_t*;
```

A `const`-qualification of the type written as:

```
const IntPtr ptr = &someValue;
```

Results in this expansion:

```
const (std::int32_t*) ptr = &someValue;
```

In this expression, `ptr` is a constant pointer, which cannot be reassigned to another memory location. However, a developer or reviewer might expect this expansion:

```
(const std::intr32_t) *ptr = &someValue;
```

In this expression, `ptr` is a pointer to a constant, which means that the contents of the location that `ptr` points to, or `*ptr`, cannot be changed.

To avoid this confusion, place a `const` or `volatile` qualifier to the right of a data type defined through `typedef` or `using`. For instance:

```
IntPtr const ptr = &someValue;
```

The only possible expansion of this expression is:

```
std::intr32_t const *ptr = &someValue;
```

which makes `ptr` a constant pointer.

Polyspace Implementation

The checker flags situations where `const` or `volatile` qualifiers are placed on the left side of data types defined through `typedef` or `using` statements.

The checker flags both pointer and nonpointer data types.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A7-1-4

The register keyword shall not be used.

Description

Rule Definition

The register keyword shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A7-1-6

The typedef specifier shall not be used

Description

Rule Definition

The typedef specifier shall not be used.

Rationale

The `using` syntax is a better alternative to `typedef`-s for defining aliases.

Since C++11, the `using` syntax allows you to define template aliases where the template arguments are not bound to a data type. For instance, the following statements define an alias `vectorType` for `vector`, where the argument `T` is not bound to a data type and can be substituted later:

```
template<class T, class Allocator = allocator<T>> class vector;
template<class T> using vectorType = vector<T, My_allocator<T>>;
vectorType<int> primes = {2,3,5,7,11,13,17,19,23,29};
```

The `typedef` keyword does not allow defining such template aliases.

Polyspace Implementation

The rule checker flags all uses of the `typedef` keyword.

If you do not want to remove certain instances of the `typedef` keyword, add a comment justifying those results. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `typedef` Keyword

```
#include <cstdint>
#include <type_traits>

typedef std::int32_t (*fptr1) (std::int32_t); //Noncompliant
using fptr2 = std::int32_t (*) (std::int32_t); //Compliant

template <class T> using fptr3 = std::int32_t (*) (T); //Compliant
```

The alias definitions for `fptr1` and `fptr2` are exactly equivalent. There is no `typedef` equivalent for the alias definition for `fptr3`.

The use of `typedef`-s violates this rule. The rule requires that you stick to the `using` syntax for consistency even when a `typedef` equivalent exists.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A7-1-7

Each expression statement and identifier declaration shall be placed on a separate line.

Description

Rule Definition

Each expression statement and identifier declaration shall be placed on a separate line.

Polyspace Implementation

The checker raises a violation when two consecutive expression statements are on the same line (unless the statements are part of a macro definition).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A7-1-8

A non-type specifier shall be placed before a type specifier in a declaration.

Description

Rule Definition

A non-type specifier shall be placed before a type specifier in a declaration.

Rationale

Non-type specifiers include:

- `typedef`.
- `friend`.
- `constexpr`.
- `register`.
- `static`.
- `extern`.
- `thread-local`.
- `mutable`.
- `inline`.
- `virtual`.
- `explicit`.

To make the code more readable, place non-type specifiers before type specifiers in a declaration.

Polyspace Implementation

Polyspace flags declarations that place non-type specifiers after a type specifier. If more than one non-type specifiers follow a type specifier, Polyspace flags the rightmost non-type specifier.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Place Non-Type Specifiers Before Type Specifiers

The following example demonstrate the Polyspace implementation of AUTOSAR rule A7-1-8.

```
#include <cstdint>

typedef std::int32_t int1;  // Compliant
std::int32_t typedef int2;  // Noncompliant

class to_be_friend {
```

```
    explicit to_be_friend(int); // Compliant
    static void *foo(void);     // Compliant
    void static *bar(void);     // Noncompliant
    virtual inline void i1(void) {}; // Compliant
    inline void virtual i2(void) {}; // Noncompliant
    constexpr register long long l1 = 0; // Compliant
    register  long long static constexpr l3 = 0; //Noncompliant
};
main(){
    //...
}
```

Polyspace flags declarations where you place non-type specifiers after type-specifiers. The declaration of the static object `l3` is flagged because the non-type specifiers `static` and `constexpr` are placed after the type-specifier `long long`. The violation is highlighted on the rightmost non-type specifier, which is `constexpr`.

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A7-1-9

A class, structure, or enumeration shall not be declared in the definition of its type.

Description

Rule Definition

A class, structure, or enumeration shall not be declared in the definition of its type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A7-2-2

Enumeration underlying type shall be explicitly defined.

Description

Rule Definition

Enumeration underlying type shall be explicitly defined.

Rationale

In an unscoped enumeration declaration such as:

`enum someEnum : type { ... }`

if `: type` is omitted, the underlying type is implementation-defined (with the only requirement that the type must accommodate all the enumeration values). Not declaring an underlying type explicitly results in implementation-defined behavior.

In a scoped enumeration declaration such as:

`enum class someEnum : type { ... }`

if `: type` is omitted, the underlying type is `int`. If an enumeration value exceeds the values allowed for `int`, you see compilation errors.

For both unscoped and scoped enumerations, declare the underlying type explicitly to avoid implementation-defined behavior or compilation errors.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Enums with Underlying Type Omitted

```
#include <cstdint>

enum E1 { //Noncompliant unscoped enum
    E10,
    E11,
    E12
};

enum E2 : std::uint8_t { //Compliant unscoped enum
    E20,
    E21,
    E22
};

enum class E3 { //Noncompliant scoped enum
    E30,
```

```
    E31,
    E32
};

enum class E4 : std::uint8_t { //Compliant scoped enum
    E40,
    E41,
    E42
};
```

In this example, the code is noncompliant when the underlying types of the enumerations are omitted.

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019b

AUTOSAR C++14 Rule A7-2-3

Enumerations shall be declared as scoped enum classes.

Description

Rule Definition

Enumerations shall be declared as scoped enum classes.

Rationale

Enumeration values in an unscoped enum can conflict with other identifiers in the same scope as the enum and cause compilation errors. For instance:

```
enum E: std::int32_t { E0, E1};
std::int32_t E0;
```

If you scope the enum, such conflicts can be avoided. For instance:

```
enum class E: std::int32_t { E0, E1};
std::int32_t E0;
```

Scoping the enum also disallows implicit conversions of the enumeration values to other types.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Unscoped Enums

```
#include<cstdint>

enum E1: std::int32_t { E10, E11}; //Noncompliant
// std::int32_t E10; causes compilation errors

enum class E2: std::int32_t { E20, E21}; //Compliant
std::int32_t E20;
```

In this example, the declaration of unscoped enum `E1` is noncompliant. Redeclaring an enumeration value of the unscoped enum causes compilation errors (as shown in the commented line that redeclares the enumeration value `E10`).

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019b

AUTOSAR C++14 Rule A7-2-4

In an enumeration, either (1) none, (2) the first or (3) all enumerators shall be initialized.

Description

Rule Definition

In an enumeration, either (1) none, (2) the first or (3) all enumerators shall be initialized.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A7-3-1

All overloads of a function shall be visible from where it is called.

Description

Rule Definition

All overloads of a function shall be visible from where it is called.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A7-4-1

The asm declaration shall not be used.

Description

Rule Definition

The asm declaration shall not be used.

Rationale

The `asm` declaration is a method to include assembly instructions directly within C++ source code. Support and implementation of the `asm` declaration is inconsistent across environments. The `asm` declaration interacts differently with C++ source code in different environments. To avoid restricting the portability of your code, do not use the `asm` declaration and do not include assembly instructions in your C++ source code.

Polyspace Implementation

Polyspace flags the use of the `asm` declaration anywhere in C++ source code.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Avoid Assembly Instructions in C++ Source Code

```
#include <cstdint>
using namespace std;
const char* p = "hello world";

void Fn1(void)
{
    asm("movq p, %rdi\n"
        "call puts");    // Noncompliant
}

#define _debug() asm volatile("debug":::"memory")
                              // Noncompliant
void Fn2(void)
{
    _debug();
}

main()
{
     //
}
```

Polyspace flags the use of `asm` declaration in `Fn(1)` because the assembly instructions following the declaration are environment-specific. For example, if you use a gcc compiler in a x64 Linux environment, `Fn1()` produces the string `hello world` when called. In other environments, the output of the call to `Fn1()` is unpredictable. Polyspace also flags the use of the `asm` declaration in creating the `_debug()` macro.

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A7-5-1

A function shall not return a reference or a pointer to a parameter that is passed by reference to const.

Description

Rule Definition

A function shall not return a reference or a pointer to a parameter that is passed by reference to const.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A7-5-2

Functions shall not call themselves, either directly or indirectly.

Description

Rule Definition

Functions shall not call themselves, either directly or indirectly.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A8-2-1

When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters.

Description

Rule Definition

When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters.

Rationale

When the return type of a template depends on the types of parameters, using the trailing return type syntax improves readability of the code significantly compared to alternatives.

For instance, for out-of-class definitions of methods, using the trailing return type syntax saves you from having to use the fully qualified return type of a function along with the `typename` keyword. For instance, instead of explicitly specifying the fully qualified return type for `aMethod` in this example:

```
template <typename T>
class aClass {
    public:
      using vectorType = std::vector<T>;
      vectorType aMethod(T const&);
};

//Difficult-to-read method definition
//Part in bold indicates fully qualified return type of method
template <typename T>
typename aClass<T>::vectorType aClass<T>::aMethod(T const &) {
};
```

You can use the trailing return type syntax as follows:

```
template <typename T>
class aClass {
    public:
      using vectorType = std::vector<T>;
      vectorType aMethod(T const&);
};
template <typename T>
auto aClass<T>::aMethod(T const &) -> vectorType {
};
```

Polyspace Implementation

The checker flags function template declarations where the explicitly specified return type of a template function has the same scope as the template function itself .

For instance, in the preceding example, the function `aMethod` has a return type `vectorType`, which has the same scope as `aMethod`, namely the class `aClass<T>`. Instead of explicitly specifying the fully qualified return type, you can use the trailing return type syntax.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A8-4-1

Functions shall not be defined using the ellipsis notation.

Description

Rule Definition

Functions shall not be defined using the ellipsis notation.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A8-4-2

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Description

Rule Definition

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A8-4-7

"in" parameters for "cheap to copy" types shall be passed by value.

Description

Rule Definition

"*in*" *parameters for* "*cheap to copy*" *types shall be passed by value.*

Polyspace Implementation

Report constant parameters references with `sizeof <= 2 * sizeof(int)`. Does not report for copy-constructor.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A8-5-0

All memory shall be initialized before it is read.

Description

Rule Definition

All memory shall be initialized before it is read.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A8-5-1

In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition.

Description

Rule Definition

In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A8-5-2

Braced-initialization {}, without equals sign, shall be used for variable initialization.

Description

Rule Definition

Braced-initialization {}, without equals sign, shall be used for variable initialization.

Rationale

Braced initialization:

```
classType Object{arg1, arg2, ...};
```

is less ambiguous than other forms of initialization. Braced initialization has the following advantages:

- Prevents implicit narrowing conversions such as from `double` to `float`.
- Avoids the ambiguous syntax that leads to the problem of most vexing parse.

  For instance, from the declaration:

  ```
  ResourceType aResource();
  ```

  It is not immediately clear if `aResource` is a function returning a variable of type `ResourceType` or an object of type `ResourceType`.

  For more information, see `Ambiguous declaration syntax`.

The rule also forbids the use of = sign for initialization because the = sign can give the impression that an assignment or copy constructor is invoked even though it is not.

Polyspace Implementation

In general, the checker flags initializations of an object `obj1` of data type `Type` using these formats:

- `Type obj1 = obj2;`
- `Type obj1(obj2);`

Provided `obj1` and `obj2` have distinct data types.

The checker is enabled only if you specify a C++ version of C++11 or later. See `C++ standard version (-cpp-version)`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Braced and Nonbraced Initialization

```
class ResourceType {
      int memberOne;
      int memberTwo;
    public:
      ResourceType() {memberOne = 0; memberTwo = 0;}
      ResourceType(int m, int n) {memberOne = m; memberTwo = n;}
      ResourceType(ResourceType &anotherResource) {
          memberOne = anotherResource.memberTwo;
          memberTwo = anotherResource.memberOne;
      }
};

void func() {
    ResourceType aResourceOne(); //Noncompliant
    ResourceType aResourceTwo(1, 2); //Noncompliant
    ResourceType aResourceThree = {1,2};   //Noncompliant

    ResourceType aResourceFour{1,2}; //Compliant

}
```

In this example, the function `func` declares four objects of type `ResourceType`. Only the declaration of `aResourceFour` does not violate this rule.

The declarations of `aResourceOne`, `aResourceTwo` and `aResourceThree` violate the rule. In particular:

- The declaration of `aResourceOne` suffers from the problem of most vexing parse. It is not clear whether `aResourceOne` is an object of type `ResourceType` or a function returning an object of type `ResourceType`.
- The declaration of `aResourceThree` seems to suggest that the copy constructor `ResourceType(ResourceType &)` is invoked for initialization. The copy constructor initializes the data member `memberOne` to 2 and `memberTwo` to 1. However, the constructor `ResourceType(int, int)` is invoked. This constructor initializes the data member `memberOne` to 1 and `memberTwo` to 2.

Check Information

Group: Declarators

See Also

`Ambiguous declaration syntax` | `Check AUTOSAR C++ 14 (-autosar-cpp14)` | `Improper array initialization` | `Non-initialized variable` | `Variable shadowing` | `Write without a further read`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A8-5-3

A variable of type auto shall not be initialized using {} or ={} braced-initialization.

Description

Rule Definition

A variable of type auto shall not be initialized using {} or ={} braced-initialization.

Rationale

Type deduction for `auto` has a counter-intuitive result when the initialization uses braces. The deduced type is `std::initializer_list<>` instead of the type that you might guess from the initializer.

For instance, the definition:

```
auto x{1};
```

results in the type of `x` being `std::initializer_list<int>` instead of `int`. Some compilers deduce an `int` type from this definition, but the behavior is not uniform across compilers.

Polyspace Implementation

The checker flags variable definitions that use the type `auto` if the variable is initialized using the `{}` or `={}` braced initialization.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `auto` in Braced Initialization

```
#include<initializer_list>

void func() {
    auto aVar{1}; //Noncompliant
    auto anotherVar(1); //Compliant
    int aThirdVar{1}; //Compliant

    auto aVarList{1,2,3}; //Noncompliant
    std::initializer_list<int> anotherVarList{1,2,3}; //Compliant
}
```

In this example, the rule is violated when the `auto` type is used with braced initialization. Instead of `auto`, an explicit type specification is preferred. Alternatively, the initialization can use parenthesis `()`, which ensures the expected type deduction.

Check Information

Group: Declarators

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2020a

AUTOSAR C++14 Rule A9-3-1

Member functions shall not return non-const "raw" pointers or references to private or protected data owned by the class.

Description

Rule Definition

Member functions shall not return non-const "raw" pointers or references to private or protected data owned by the class.

Polyspace Implementation

The checker flags a rule violation only if a member function returns a non-`const` pointer or reference to a nonstatic data member. The rule does not apply to static data members.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Classes

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A9-5-1

Unions shall not be used.

Description

Rule Definition

Unions shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Classes

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule A9-6-1

Bit-fields shall be either unsigned integral, or enumeration (with underlying type of unsigned integral type).

Description

Rule Definition

Bit-fields shall be either unsigned integral, or enumeration (with underlying type of unsigned integral type).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M0-1-1

A project shall not contain unreachable code.

Description

Rule Definition

A project shall not contain unreachable code.

Rationale

This rule flags situations where a group of statements is unreachable because of syntactic reasons. For instance, code following a `return` statement are always unreachable.

Unreachable code involve unnecessary maintenance and can often indicate programming errors.

Polyspace Implementation

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Unreachable statements

```
int func(int arg) {
 int temp = 0;
 switch(arg) {
     temp = arg; // Noncompliant
     case 1:
     {
         break;
     }
     default:
     {
         break;
     }
 }
 return arg;
 arg++; // Noncompliant
}
```

These statements are unreachable:

- Statements inside a `switch` statement that do not belong to a `case` or `default` block.
- Statements after a `return` statement.

Check Information

Group: Language Independent Issues

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M0-1-10

Every defined function should be called at least once.

Description

Rule Definition

Every defined function should be called at least once.

Rationale

If a function with a definition is not called, it might indicate a serious coding error. For instance, the function call is unreachable or a different function is called unintentionally.

Polyspace Implementation

The checker detects situations where a static function is defined but not called at all in its translation unit.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Uncalled Static Function

```
static void func1() {
}

static void func2() { //Noncompliant
}

void func3();

int main() {
    func1();
    return 0;
}
```

The `static` function `func2` is defined but not called.

The function `func3` is not called either, however, it is only declared and not defined. The absence of a call to `func3` does not violate the rule.

Check Information

Group: Language Independent Issues

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M0-1-2

A project shall not contain infeasible paths.

Description

Rule Definition

A project shall not contain infeasible paths.

Rationale

This rule flags situations where a group of statements is redundant because of nonsyntactic reasons. For instance, an `if` condition is always true or false. Code that is unreachable from syntactic reasons are flagged by rule 0-1-1.

Unreachable or redundant code involve unnecessary maintenance and can often indicate programming errors.

Polyspace Implementation

Bug Finder and Code Prover check this rule differently. The analysis can produce different results.

- Bug Finder uses the `Dead code` and `Useless if` checkers to detect violations of this rule.
- Code Prover does not use run-time checks to detect violations of this rule. Instead, Code Prover detects the violations at compile time.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Boolean Operations with Invariant Results

```
void func (unsigned int arg) {
 if (arg >= 0U) //Noncompliant
     arg  = 1U;
 if (arg < 0U) //Noncompliant
     arg = 1U;
}
```

An `unsigned int` variable is nonnegative. Both `if` conditions involving the variable are always true or always false and are therefore redundant.

Check Information

Group: Language Independent Issues

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M0-1-3

A project shall not contain unused variables.

Description

Rule Definition

A project shall not contain unused variables.

Polyspace Implementation

The checker flags local or global variables that are declared or defined but not used anywhere in the source files. This specification also applies to members of structures and classes.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of Named Bit Field for Padding

```
#include <iostream>
struct S {
    unsigned char b1 : 3;
    unsigned char pad: 1;  //Noncompliant
    unsigned char b2 : 4;
};
void init(struct S S_obj)
{
    S_obj.b1 = 0;
    S_obj.b2 = 0;
}
```

In this example, the bit field `pad` is used for padding the structure. Therefore, the field is never read or written and causes a violation of this rule. To avoid the violation, use an unnamed field for padding.

```
struct S {
    unsigned char b1 : 3;
    unsigned char : 1;
    unsigned char b2 : 4;
};
```

Check Information

Group: Language Independent Issues

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M0-1-9

There shall be no dead code.

Description

Rule Definition

There shall be no dead code.

Rationale

If an operation is reachable but removing the operation does not affect program behavior, the operation constitutes dead code. For instance, suppose that a variable is never read following a write operation. The write operation is redundant.

The presence of dead code can indicate an error in the program logic. Because a compiler can remove dead code, its presence can cause confusion for code reviewers.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Redundant Operations

```
#define ULIM 10000

int func(int arg) {
    int res;
    res = arg*arg + arg;
    if (res > ULIM)
        res = 0; //Noncompliant
    return arg;
}
```

In this example, the operations involving `res` are redundant because the function `func` returns its argument `arg`. All operations involving `res` can be removed without changing the effect of the function.

The checker flags the last write operation on `res` because the variable is never read after that point. The dead code can indicate an unintended coding error. For instance, you intended to return the value of `res` instead of `arg`.

Check Information

Group: Language Independent Issues

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M0-2-1

An object shall not be assigned to an overlapping object.

Description

Rule Definition

An object shall not be assigned to an overlapping object.

Rationale

When you assign an object to another object with overlapping memory, the behavior is undefined.

The exceptions are:

- You assign an object to another object with exactly overlapping memory and compatible type.
- You copy one object to another with `memmove`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Assignment of Union Members

```
void func (void) {
    union {
        short i;
        int j;
    } a = {0}, b = {1};

    a.j = a.i;   //Noncompliant
    a = b;       //Compliant
}
```

In this example, the rule is violated when `a.i` is assigned to `a.j` because the two variables have overlapping regions of memory.

Check Information

Group: Language Independent Issues

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M10-1-1

Classes should not be derived from virtual bases.

Description

Rule Definition

Classes should not be derived from virtual bases.

Rationale

The use of virtual bases can lead to many confusing behaviors.

For instance, in an inheritance hierarchy involving a virtual base, the most derived class calls the constructor of the virtual base. Intermediate calls to the virtual base constructor are ignored.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of Virtual Bases

```
class Base {};
class Intermediate: public virtual Base {}; //Noncompliant
class Final: public Intermediate {};
```

In this example, the rule checker raises a violation when the `Intermediate` class is derived from the class `Base` with the `virtual` keyword.

The following behavior can be a potential source of confusion. When you create an object of type `Final`, the constructor of `Final` directly calls the constructor of `Base`. Any call to the `Base` constructor from the `Intermediate` constructor are ignored. You might see unexpected results if you do not take into account this behavior.

Check Information

Group: Derived Classes

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M10-1-2

A base class shall only be declared virtual if it is used in a diamond hierarchy.

Description

Rule Definition

A base class shall only be declared virtual if it is used in a diamond hierarchy.

Rationale

This rule is less restrictive than `AUTOSAR C++14 Rule M10-1-1`. Rule M10-1-1 forbids the use of a virtual base anywhere in your code because a virtual base can lead to potentially confusing behavior.

Rule M10-1-2 allows the use of virtual bases in the one situation where they are useful, that is, as a common base class in diamond hierarchies.

For instance, the following diamond hierarchy violates rule M10-1-1 but not rule M10-1-2.

```
class Base {};
class Intermediate1: public virtual Base {};
class Intermediate2: public virtual Base {};
class Final: public Intermediate1, public Intermediate2 {};
```

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Derived Classes

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M10-1-3

An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

Description

Rule Definition

An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

Rationale

The checker flags situations where the same class is inherited as a virtual base class and a non-virtual base class in the same derived class. These situations defeat the purpose of virtual inheritance and causes multiple copies of the base class sub-object in the derived class object.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Base Class Both Virtual and Non-Virtual in Same Hierarchy

```
class Base {};
class Intermediate1: virtual public Base {};
class Intermediate2: virtual public Base {};
class Intermediate3: public Base {};
class Final: public Intermediate1, Intermediate2, Intermediate3 {}; //Noncompliant
```

In this example, the class `Base` is inherited in `Final` both as a virtual and non-virtual base class. The `Final` object contains at least two copies of a `Base` sub-object.

Check Information

Group: Derived Classes

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M10-2-1

All accessible entity names within a multiple inheritance hierarchy should be unique.

Description

Rule Definition

All accessible entity names within a multiple inheritance hierarchy should be unique.

Polyspace Implementation

The checker flags data members from different classes with conflicting names if the same class derives from these classes. For instance:

```
class B1
  {
    public:
      int count;
      void foo ( );
  };
class B2
  {
    public:
      int count;
      void foo ( );
  };

class D : public B1, public B2
  {
    public:
      void Bar ( )
      {
        ++B1::count;
        B1::foo ( );
      }
  };
```

If the data member access in the derived class is ambiguous, the analysis reports this issue as a compilation error and not a coding rule violation. For instance, a compilation error occurs in the preceding example if the class `D` is rewritten as:

```
class D : public B1, public B2
  {
    public:
      void Bar ( )
      {
        ++count;      // Is that B1::count or B2::count?
        foo ( );      // Is that B1::foo() or B2::foo()?
      }
  };
```

The checker does not check for conflicts between entities of different kinds, for instance, member functions against data members.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Derived Classes

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M10-3-3

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

Description

Rule Definition

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Derived Classes

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M11-0-1

Member data in non-POD class types shall be private.

Description

Rule Definition

Member data in non-POD class types shall be private.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Member Access Control

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M12-1-1

An object's dynamic type shall not be used from the body of its constructor or destructor.

Description

Rule Definition

An object's dynamic type shall not be used from the body of its constructor or destructor.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Special Member Functions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M14-5-3

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.

Description

Rule Definition

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Templates

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M14-6-1

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.

Description

Rule Definition

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Templates

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M15-0-3

Control shall not be transferred into a try or catch block using a goto or a switch statement.

Description

Rule Definition

Control shall not be transferred into a try or catch block using a goto or a switch statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M15-1-2

NULL shall not be thrown explicitly.

Description

Rule Definition

NULL shall not be thrown explicitly.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M15-1-3

An empty throw (throw;) shall only be used in the compound statement of a catch handler.

Description

Rule Definition

An empty throw (throw;) shall only be used in the compound statement of a catch handler.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M15-3-3

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

Description

Rule Definition

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M15-3-6

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.

Description

Rule Definition

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M15-3-7

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.

Description

Rule Definition

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Exception Handling

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M16-0-1

#include directives in a file shall only be preceded by other pre-processor directives or comments.

Description

Rule Definition

#include directives in a file shall only be preceded by other pre-processor directives or comments.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M16-0-2

Macros shall only be #define'd or #undef'd in the global namespace.

Description

Rule Definition

Macros shall only be #define'd or #undef'd in the global namespace.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M16-0-5

Arguments to a function-like macro shall not contain tokens that look like pre-processing directives.

Description

Rule Definition

Arguments to a function-like macro shall not contain tokens that look like pre-processing directives.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M16-0-6

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.

Description

Rule Definition

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M16-0-7

Undefined macro identifiers shall not be used in #if or #elif pre-processor directives, except as operands to the defined operator.

Description

Rule Definition

Undefined macro identifiers shall not be used in #if or #elif pre-processor directives, except as operands to the defined operator.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M16-0-8

If the # token appears as the first token on a line, then it shall be immediately followed by a pre-processing token.

Description

Rule Definition

If the # token appears as the first token on a line, then it shall be immediately followed by a pre-processing token.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M16-1-1

The defined pre-processor operator shall only be used in one of the two standard forms.

Description

Rule Definition

The defined pre-processor operator shall only be used in one of the two standard forms.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M16-1-2

All #else, #elif and #endif pre-processor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

Description

Rule Definition

All #else, #elif and #endif pre-processor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M16-2-3

Include guards shall be provided.

Description

Rule Definition

Include guards shall be provided.

Polyspace Implementation

The checker raises a violation if a header file does not contain an include guard.

For instance, this code uses an include guard for the `#define` and `#include` statements and does not violate the rule:

```
// Contents of a header file
#ifndef FILE_H

#define FILE_H
#include "libFile.h"

#endif
```

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M16-3-1

There shall be at most one occurrence of the # or ## operators in a single macro definition.

Description

Rule Definition

There shall be at most one occurrence of the # or ## operators in a single macro definition.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M16-3-2

The # and ## operators should not be used.

Description

Rule Definition

The # and ## operators should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Preprocessing Directives

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M17-0-2

The names of standard library macros and objects shall not be reused.

Description

Rule Definition

The names of standard library macros and objects shall not be reused.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Library Introduction

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M17-0-3

The names of standard library functions shall not be overridden.

Description

Rule Definition

The names of standard library functions shall not be overridden.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Library Introduction

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M17-0-5

The setjmp macro and the longjmp function shall not be used.

Description

Rule Definition

The setjmp macro and the longjmp function shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Library Introduction

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M18-0-3

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

Description

Rule Definition

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language Support Library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M18-0-4

The time handling functions of library <ctime> shall not be used.

Description

Rule Definition

The time handling functions of library <ctime> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language Support Library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M18-0-5

The unbounded functions of library <cstring> shall not be used.

Description

Rule Definition

The unbounded functions of library <cstring> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language Support Library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M18-2-1

The macro offsetof shall not be used.

Description

Rule Definition

The macro offsetof shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language Support Library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M18-7-1

The signal handling facilities of <csignal> shall not be used.

Description

Rule Definition

The signal handling facilities of <csignal> shall not be used.

Rationale

Signal handling functions such as `signal` contains undefined and implementation-specific behavior.

You have to be very careful when using `signal` to avoid these behaviors.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Language Support Library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M19-3-1

The error indicator errno shall not be used.

Description

Rule Definition

The error indicator errno shall not be used.

Rationale

Observing this rule encourages the good practice of not relying on `errno` to check error conditions.

Checking `errno` is not sufficient to guarantee absence of errors. Functions such as `fopen` might not set `errno` on error conditions. Often, you have to check the return value of such functions for error conditions.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `errno`

```
#include <cstdlib>
#include <cerrno>

void func (const char* str) {
    errno = 0;  // Noncompliant
    int i = atoi(str);
    if(errno != 0) { // Noncompliant
        //Handle Error
    }
}
```

The use of `errno` violates this rule. The function `atoi` is not required to set `errno` if the input string cannot be converted to an integer. Checking `errno` later does not safeguard against possible failures in conversion.

Check Information

Group: Diagnostics Library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M2-10-1

Different identifiers shall be typographically unambiguous.

Description

Rule Definition

Different identifiers shall be typographically unambiguous.

Rationale

When you use identifiers that are typographically close, you can confuse between them.

The identifiers should not differ by:

- The interchange of a lowercase letter with its uppercase equivalent.
- The presence or absence of the underscore character.
- The interchange of the letter `O` and the digit `0`.
- The interchange of the letter `I` and the digit `1`.
- The interchange of the letter `I` and the letter `l`.
- The interchange of the letter `S` and the digit `5`.
- The interchange of the letter `Z` and the digit `2`.
- The interchange of the letter `n` and the letter `h`.
- The interchange of the letter `B` and the digit `8`.
- The interchange of the letters `rn` and the letter `m`.

Polyspace Implementation

The rule checker does not consider the fully qualified names of variables when checking this rule.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Typographically Ambiguous Identifiers

```
void func(void) {
    int id1_numval;
    int id1_num_val;  /* Non-compliant */

    int id2_numval;
    int id2_numVal;   /* Non-compliant */
```

```
    int id3_lvalue;
    int id3_Ivalue;   /* Non-compliant */

    int id4_xyz;
    int id4_xy2;      /* Non-compliant */

    int id5_zerO;
    int id5_zer0;     /* Non-compliant */

    int id6_rn;
    int id6_m;        /* Non-compliant */
}
```

In this example, the rule is violated when identifiers that can be confused for each other are used.

Check Information

Group: Lexical Conventions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M2-13-2

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.

Description

Rule Definition

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.

Rationale

Octal constants are denoted by a leading zero. A developer or code reviewer can mistake an octal constant as a decimal constant with a redundant leading zero.

Octal escape sequences beginning with \ can also cause confusion. Inadvertently introducing an 8 or 9 in the digit sequence after \ breaks the escape sequence and introduces a new digit. A developer or code reviewer can ignore this issue and continue to treat the escape sequence as one digit.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of Octal Constants and Octal Escape Sequences

```
void func(void) {
  int busData[6];

  busData[0] = 100;
  busData[1] = 108;
  busData[2] = 052;      //Noncompliant
  busData[3] = 071;      //Noncompliant
  busData[4] = '\109';   //Noncompliant
  busData[5] = '\100';   //Noncompliant

}
```

The checker flags all octal constants (other than zero) and all octal escape sequences (other than `\0`).

In this example:

- The octal escape sequence contains the digit 9, which is not an octal digit. This escape sequence has implementation-defined behavior.
- The octal escape sequence `\100` represents the number 64, but the rule checker forbids this use.

Check Information

Group: Lexical Conventions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M2-13-3

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

Description

Rule Definition

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

Rationale

The signedness of a constant is determined from:

- Value of the constant.
- Base of the constant: octal, decimal or hexadecimal.
- Size of the various types.
- Any suffixes used.

Unless you use a suffix `u` or `U`, another developer looking at your code cannot determine easily whether a constant is signed or unsigned.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Lexical Conventions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M2-13-4

Literal suffixes shall be upper case.

Description

Rule Definition

Literal suffixes shall be upper case.

Rationale

Literal constants can end with the letter `l` (el). Enforcing literal suffixes to be upper case removes potential confusion between the letter `l` and the digit `1`.

For consistency, use upper case constants for other suffixes such as `U` (unsigned) and `F` (float).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of Literal Constants with Lower Case Suffix

```
const int a = 0l; //Noncompliant
const int b = 0L; //Compliant
```

In this example, both `a` and `b` are assigned the same literal constant. However, from a quick glance, one can mistakenly assume that `a` is assigned the value `01` (octal one).

Check Information

Group: Lexical Conventions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M27-0-1

The stream input/output library <cstdio> shall not be used.

Description

Rule Definition

The stream input/output library <cstdio> shall not be used.

Rationale

Functions in `cstdio` such as `gets`, `fgetpos`, `fopen`, `ftell`, etc. have unspecified, undefined and implementation-defined behavior.

For instance:

- The `gets` function:

  ```
  char * gets ( char * buf );
  ```

  does not check if the number of characters provided at the standard input exceeds the buffer `buf`. The function can have unexpected behavior when the input exceeds the buffer.
- The `fopen` function has implementation-specific behavior related to whether it sets `errno` on errors or whether it accepts additional characters following the standard mode specifiers.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of `gets`

```
#include <cstdio>

void func() {
 char array[10];
 gets(array);
}
```

The use of `gets` violates this rule.

Check Information

Group: Input Output Library

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M2-7-1

The character sequence /* shall not be used within a C-style comment.

Description

Rule Definition

The character sequence / shall not be used within a C-style comment.*

Rationale

If your code contains a `/*` in a `/* */` comment, it typically means that you have inadvertently commented out code. See the example that follows.

Polyspace Implementation

You cannot justify a violation of this rule using source code annotations.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of /* in /* */ Comment

```
void foo() {
    /* Initializer functions
     setup();
    /* Step functions */
}
```

In this example, the call to `setup()` is commented out because the ending `*/` is omitted, perhaps inadvertently. The checker flags this issue by highlighting the `/*` in the `/* */` comment.

Check Information

Group: Lexical Conventions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M3-1-2

Functions shall not be declared at block scope.

Description

Rule Definition

Functions shall not be declared at block scope.

Rationale

It is a good practice to place all declarations at the namespace level.

Additionally, if you declare a function at block scope, it is often not clear if the statement is a function declaration or an object declaration with a call to the constructor.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Function Declarations at Block Scope

```
class A {
};

void b1() {
    void func(); //Noncompliant
    A a();   //Noncompliant
}
```

In this example, the declarations of `func` and `a` are in the block scope of `b1`.

The second function declaration can cause confusion because it is not clear if `a` is a function that returns an object of type `A` or `a` is itself an object of type `A`.

Check Information

Group: Basic Concepts

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M3-2-1

All declarations of an object or function shall have compatible types.

Description

Rule Definition

All declarations of an object or function shall have compatible types.

Rationale

If the declarations of an object or function in two different translation units have incompatible types, the behavior is undefined.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Basic Concepts

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M3-2-2

The One Definition Rule shall not be violated.

Description

Rule Definition

The One Definition Rule shall not be violated.

Rationale

Violations of the One Definition Rule leads to undefined behavior.

Polyspace Implementation

The checker flags situations where the same function or object has multiple definitions and the definitions differ by some token.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Different Tokens in Same Type Definition

This example uses two files:

- `file1.cpp`:

```
struct S
{
   int x;
   int y;
};
```

- `file2.cpp`:

```
struct S
{
   int y;
   int x;
};
```

In this example, both `file1.cpp` and `file2.cpp` define the structure `S`. However, the definitions switch the order of the structure fields.

Check Information

Group: Basic Concepts

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M3-2-3

A type, object or function that is used in multiple translation units shall be declared in one and only one file.

Description

Rule Definition

A type, object or function that is used in multiple translation units shall be declared in one and only one file.

Rationale

If you declare an identifier in a header file, you can include the header file in any translation unit where the identifier is defined or used. In this way, you ensure consistency between:

- The declaration and the definition.
- The declarations in different translation units.

The rule enforces the practice of declaring external objects or functions in header files.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Basic Concepts

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M3-2-4

An identifier with external linkage shall have exactly one definition.

Description

Rule Definition

An identifier with external linkage shall have exactly one definition.

Rationale

If an identifier has multiple definitions or no definitions, it can lead to undefined behavior.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Multiple Definitions of Identifier

This example uses two files:

- `file1.cpp`:

  ```
  int x = 0;
  ```

- `file2.cpp`:

  ```
  int x = 1;
  ```

The same identifier `x` is defined in both files.

Check Information

Group: Basic Concepts

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M3-3-2

If a function has internal linkage then all re-declarations shall include the static storage class specifier.

Description

Rule Definition

If a function has internal linkage then all re-declarations shall include the static storage class specifier.

Rationale

If a function declaration has the `static` storage class specifier, it has internal linkage. Subsequent redeclarations of the function have internal linkage even without the `static` specifier.

However, if you do not specify the `static` keyword explicitly, it is not immediately clear from a declaration whether the function has internal linkage.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Missing `static` Specifier from Redeclaration

```
static void func1 ();
static void func2 ();

void func1() {}  //Noncompliant
static void func2() {}
```

In this example, the function `func1` is declared `static` but defined without the `static` specifier.

Check Information

Group: Basic Concepts

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M3-4-1

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

Description

Rule Definition

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

Rationale

Defining variables with the minimum possible block scope reduces the possibility that they might later be accessed unintentionally.

For instance, if an object is meant to be accessed in one function only, declare the object local to the function.

Polyspace Implementation

The rule checker determines if an object is used in one block only. If the object is used in one block but defined outside the block, the checker raises a violation.

When you declare a variable outside a range-based `for` loop and use it only inside the loop block, Polyspace flags the variable. If you cannot declare the variable inside the loop block, justify this result using comments in your result or code. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Use of Global Variable in Single Function

```
static int countReset; //Noncompliant

volatile int check;

void increaseCount() {
    int count = countReset;
    while(check%2) {
     count++;
    }
}
```

In this example, the variable `countReset` is declared global used in one function only. A compliant solution declares the variable local to the function to reduce its visibility.

Check Information

Group: Basic Concepts

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M3-9-1

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

Description

Rule Definition

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

Rationale

If a redeclaration is not token-for-token identical to the previous declaration, it is not clear from visual inspection which object or function is being redeclared.

Polyspace Implementation

The rule checker compares the current declaration with the last seen declaration.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Identical Declarations That Do Not Match Token for Token

```
typedef int* intptr;

int* map;
extern intptr map; //Noncompliant

intptr table;
extern intptr table; //Compliant
```

In this example, the variable `map` is declared twice. The second declaration uses a `typedef` which resolves to the type of the first declaration. Because of the `typedef`, the second declaration is not token-for-token identical to the first.

Check Information

Group: Basic Concepts

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M3-9-3

The underlying bit representations of floating-point values shall not be used.

Description

Rule Definition

The underlying bit representations of floating-point values shall not be used.

Rationale

The underlying bit representations of floating point values vary across compilers. If you directly use the underlying representation of floating point values, your program is not portable across implementations.

Polyspace Implementation

The rule checker flags conversions from pointers to floating point types into pointers to integer types, and vice versa.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Using Underlying Representation of Floating-Point Values

```
float fabs2(float f) {
    unsigned int* ptr = reinterpret_cast <unsigned int*> (&f); //Noncompliant
    *(ptr + 3) &= 0x7f;
    return f;
}
```

In this example, the `reinterpret_cast` attempts to cast a floating-point value to an integer and access the underlying bit representation of the floating point value.

Check Information

Group: Basic Concepts

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M4-10-1

NULL shall not be used as an integer value.

Description

Rule Definition

NULL shall not be used as an integer value.

Rationale

In C++, you can use the literals 0 and NULL as both an integer and a null pointer constant. However, use of 0 as a null pointer constant or NULL as an integer can cause developer confusion.

This rule restricts the use of NULL to null pointer constants. `AUTOSAR C++14 Rule M4-10-2` restricts the use of the literal 0 to integers.

Polyspace Implementation

The checker flags assignment of NULL to an integer variable or binary operations involving NULL and an integer. Assignments can be direct or indirect such as passing NULL as integer argument to a function.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Compliant and Noncompliant Uses of `NULL`

```
#include <cstddef>

void checkInteger(int);
void checkPointer(int *);

void main() {
    checkInteger(NULL); //Noncompliant
    checkPointer(NULL); //Compliant
}
```

In this example, the use of `NULL` as argument to the `checkInteger` function is noncompliant because the function expects an `int` argument.

Check Information

Group: Standard Conversions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M4-10-2

Literal zero (0) shall not be used as the null-pointer-constant.

Description

Rule Definition

Literal zero (0) shall not be used as the null-pointer-constant.

Rationale

In C++, you can use the literals 0 and NULL as both an integer and a null pointer constant. However, use of 0 as a null pointer constant or NULL as an integer can cause developer confusion.

This rule restricts the use of the literal 0 to integers. `AUTOSAR C++14 Rule M4-10-1` restricts the use of NULL to null pointer constants.

Polyspace Implementation

The checker flags assignment of 0 to a pointer variable or binary operations involving 0 and a pointer. Assignments can be direct or indirect such as passing 0 as pointer argument to a function.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Compliant and Noncompliant Uses of Literal 0

```
#include <cstddef>

void checkInteger(int);
void checkPointer(int *);

void main() {
    checkInteger(0); //Compliant
    checkPointer(0); //Noncompliant
}
```

In this example, the use of 0 as argument to the `checkPointer` function is noncompliant because the function expects an `int *` argument.

Check Information

Group: Standard Conversions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M4-5-1

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and ! =, the unary & operator, and the conditional operator.

Description

Rule Definition

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and ! =, the unary & operator, and the conditional operator.

Rationale

Operators other than the ones mentioned in the rule do not produce meaningful results with `bool` operands. Use of `bool` operands with these operators can indicate programming errors. For instance, you intended to use the logical operator `||` but used the bitwise operator `|` instead.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Compliant and Noncompliant Uses of `bool` Operands

```
void boolOperations() {
    bool lhs = true;
    bool rhs = false;

    int res;

    if(lhs & rhs) {} //Noncompliant
    if(lhs < rhs) {} //Noncompliant
    if(~rhs) {}      //Noncompliant
    if(lhs ^ rhs) {} //Noncompliant
    if(lhs == rhs) {} //Compliant
    if(!rhs) {}       //Compliant
    res = lhs? -1:1;  //Compliant
}
```

In this example, `bool` operands do not violate the rule when used with the `==`, `!` and the `?` operators.

Check Information

Group: Standard Conversions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M4-5-3

Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and ! =, and the unary & operator.

Description

Rule Definition

Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and ! =, and the unary & operator.

Rationale

The C++03 Standard only requires that the characters `'0'` to `'9'` have consecutive values. Other characters do not have well-defined values. If you use these characters in operations other than the ones mentioned in the rule, you implicitly use their underlying values and might see unexpected results.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Compliant and Noncompliant Uses of Character Operands

```
void charManipulations (char ch) {

    char initChar = 'a'; //Compliant
    char finalChar = 'z'; //Compliant

    if(ch == initChar) {} //Compliant
    if( (ch >= initChar) && (ch <= finalChar)) {} //Noncompliant
    else if( (ch >= '0') && (ch <= '9') ) {} //Compliant by exception
}
```

In this example, character operands do not violate the rule when used with the = and == operators. Character operands can also be used with relational operators as long as the comparison is performed with the digits `'0'` to `'9'`.

Check Information

Group: Standard Conversions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-10

If the bitwise operators ~and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Description

Rule Definition

If the bitwise operators ~and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-11

The plain char type shall only be used for the storage and use of character values.

Description

Rule Definition

The plain char type shall only be used for the storage and use of character values.

Polyspace Implementation

The checker raises a violation when a value of signed or unsigned integer type is implicitly converted to the plain char type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-12

Signed char and unsigned char type shall only be used for the storage and use of numeric values.

Description

Rule Definition

Signed char and unsigned char type shall only be used for the storage and use of numeric values.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-14

The first operand of a conditional-operator shall have type bool.

Description

Rule Definition

The first operand of a conditional-operator shall have type bool.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-15

Array indexing shall be the only form of pointer arithmetic.

Description

Rule Definition

Array indexing shall be the only form of pointer arithmetic.

Polyspace Implementation

The checker flags:

- Arithmetic operations on all pointers, for instance `p+I`, `I+p` and `p-I`, where `p` is a pointer and `I` an integer..
- Array indexing on nonarray pointers.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-17

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Description

Rule Definition

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Polyspace Implementation

Use Bug Finder for this checker. The rule checker performs the same checks as `Subtraction or comparison between pointers to different arrays`. Code Prover can fail to detect some violations.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-18

>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.

Description

Rule Definition

>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.

Polyspace Implementation

Use Bug Finder for this checker. The rule checker performs the same checks as `Subtraction or comparison between pointers to different arrays`. Code Prover can fail to detect some violations.

The checker ignores casts when showing the violation on relational operator use with pointers types.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-2

Limited dependence should be placed on C++ operator precedence rules in expressions.

Description

Rule Definition

Limited dependence should be placed on C++ operator precedence rules in expressions.

Rationale

Use parentheses to clearly indicate the order of evaluation.

Depending on operator precedence can cause the following issues:

- If you or another code reviewer reviews the code, the intended order of evaluation is not immediately clear.
- It is possible that the result of the evaluation does not meet your expectations. For instance:
  - In the operation `*p++`, it is possible that you expect the dereferenced value to be incremented. However, the pointer `p` is incremented before the dereference.
  - In the operation `(x == y | z)`, it is possible that you expect `x` to be compared with `y | z`. However, the == operation happens before the `|` operation.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Evaluation Order Dependent on Operator Precedence Rules

```
#include <cstdio>

void showbits(unsigned int x) {
    for(int i = (sizeof(int) * 8) - 1; i >= 0; i--) {
       (x & 1u << i) ? putchar('1') : putchar('0'); // Noncompliant
    }
    printf("\n");
}
```

In this example, the checker flags the operation `x & 1u << i` because the statement relies on operator precedence rules for the `<<` operation to happen before the `&` operation. If this is the intended order, the operation can be rewritten as `x & (1u << i)`.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-20

Non-constant operands to a binary bitwise operator shall have the same underlying type.

Description

Rule Definition

Non-constant operands to a binary bitwise operator shall have the same underlying type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-21

Bitwise operators shall only be applied to operands of unsigned underlying type.

Description

Rule Definition

Bitwise operators shall only be applied to operands of unsigned underlying type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-3

A cvalue expression shall not be implicitly converted to a different underlying type.

Description

Rule Definition

A cvalue expression shall not be implicitly converted to a different underlying type.

Rationale

This rule ensures that the result of the expression does not overflow when converted to a different type.

Polyspace Implementation

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

The underlying data type of a cvalue expression is the widest of operand data types in the expression. For instance, if you add two variables, one of type `int8_t` (`typedef` for `char`) and another of type `int32_t` (`typedef` for `int`), the addition has underlying type `int32_t`. If you assign the sum to a variable of type `int8_t`, the rule is violated.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Implicit Conversion of Cvalue Expression

```
typedef char int8_t;
typedef signed int int32_t;

void func ( )
  {
    int32_t s32;
    int8_t s8;
    s32 = s8 + s8; //Noncompliant
    s32 = s32 + s8; //Compliant
  }
```

In this example, the rule is violated when two variables of type `int8_t` are added and the result is assigned to a variable of type `int32_t`. The underlying type of the addition does not take into account the integer promotion involved and is simply the widest of operand data types, in this case, `int8_t`.

The rule is not violated if one of the operands has type `int32_t` and the result is assigned to a variable of type `int32_t`. In this case, the underlying data type of the addition is the same as the type of the variable to which the result is assigned.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-4

An implicit integral conversion shall not change the signedness of the underlying type.

Description

Rule Definition

An implicit integral conversion shall not change the signedness of the underlying type.

Rationale

Some conversions from signed to unsigned data types can lead to implementation-defined behavior. You can see unexpected results from the conversion.

Polyspace Implementation

The checker flags implicit conversions from a signed to an unsigned integer data type or vice versa.

The checker assumes that `ptrdiff_t` is a signed integer.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Implicit Conversions that Change Signedness

```
typedef char int8_t;
typedef unsigned char uint8_t;

void func()
  {
    int8_t s8;
    uint8_t u8;

    s8 = u8; //Noncompliant
    u8 = s8 + u8; //Noncompliant
    u8 = static_cast< uint8_t > ( s8 ) + u8; //Compliant
}
```

In this example, the rule is violated when a variable with a variable with signed data type is implicitly converted to a variable with unsigned data type or vice versa. If the conversion is explicit, as in the preceding example, the rule violation does not occur.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-5

There shall be no implicit floating-integral conversions.

Description

Rule Definition

There shall be no implicit floating-integral conversions.

Polyspace Implementation

This rule takes precedence over 5-0-4 and 5-0-6 if they apply at the same time.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-6

An implicit integral or floating-point conversion shall not reduce the size of the underlying type.

Description

Rule Definition

An implicit integral or floating-point conversion shall not reduce the size of the underlying type.

Rationale

A conversion that reduces the size of the underlying type can result in loss of information.

Polyspace Implementation

If the conversion is to a narrower integer with a different sign, then rule M5-0-4 takes precedence over rule M5-0-6. Only rule M5-0-4 is shown.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-7

There shall be no explicit floating-integral conversions of a cvalue expression.

Description

Rule Definition

There shall be no explicit floating-integral conversions of a cvalue expression.

Rationale

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation. For instance, in this example, the result of an integer division is then cast to a floating-point type.

```
short num;
short den;
float res;
res= static_cast<float> (num/den);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a floating-point division because of the later cast.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Conversion of Division Result from Integer to Floating Point

```
void func() {
     short num;
     short den;
     short res_short;
     float res_float;

     res_float = static_cast<float> (num/den); //Noncompliant

     res_short = num/den;
     res_float = static_cast<float> (res_float); //Compliant

}
```

In this example, the first cast on the division result violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the expression is evaluated with an underlying type `float`.
- The second cast makes it clear that the expression is evaluated with the underlying type `short`. The result is then cast to the type `float`.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-8

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Description

Rule Definition

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Rationale

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation. For instance, in this example, the sum of two `short` operands is cast to the wider type `int`.

```
short op1;
short op2;
int res;
res= static_cast<int> (op1 + op2);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a sum with the underlying type `int` because of the later cast.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Examples

Conversion of Sum to Wider Integer Type

```
void func() {
     short op1;
     short op2;
     int res;

     res = static_cast<int> (op1 + op2); //Noncompliant
     res = static_cast<int> (op1) + op2; //Compliant

}
```

In this example, the first cast on the sum violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the sum is evaluated with an underlying type `int`.
- The second cast first converts one of the operands to `int` so that the sum is actually evaluated with the underlying type `int`.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-0-9

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

Description

Rule Definition

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-14-1

The right hand operand of a logical &&, || operators shall not contain side effects.

Description

Rule Definition

The right hand operand of a logical &&, || operators shall not contain side effects.

Polyspace Implementation

The checker does not show a warning on volatile accesses and function calls.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-18-1

The comma operator shall not be used.

Description

Rule Definition

The comma operator shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-19-1

Evaluation of constant unsigned integer expressions shall not lead to wrap-around.

Description

Rule Definition

Evaluation of constant unsigned integer expressions shall not lead to wrap-around.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-2-10

The increment (++) and decrement (--) operators shall not be mixed with other operators in an expression.

Description

Rule Definition

The increment (++) and decrement (--) operators shall not be mixed with other operators in an expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-2-11

The comma operator, && operator and the || operator shall not be overloaded.

Description

Rule Definition

The comma operator, && operator and the || operator shall not be overloaded.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-2-12

An identifier with array type passed as a function argument shall not decay to a pointer.

Description

Rule Definition

An identifier with array type passed as a function argument shall not decay to a pointer.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-2-2

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.

Description

Rule Definition

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-2-3

Casts from a base class to a derived class should not be performed on polymorphic types.

Description

Rule Definition

Casts from a base class to a derived class should not be performed on polymorphic types.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-2-6

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

Description

Rule Definition

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-2-8

An object with integer type or pointer to void type shall not be converted to an object with pointer type.

Description

Rule Definition

An object with integer type or pointer to void type shall not be converted to an object with pointer type.

Polyspace Implementation

The checker allows an exception on zero constants.

Objects with pointer type include objects with pointer-to-function type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-2-9

A cast shall not convert a pointer type to an integral type.

Description

Rule Definition

A cast shall not convert a pointer type to an integral type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-3-1

Each operand of the ! operator, the logical && or the logical || operators shall have type bool.

Description

Rule Definition

Each operand of the ! operator, the logical && or the logical || operators shall have type bool.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-3-2

The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Description

Rule Definition

The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-3-3

The unary & operator shall not be overloaded.

Description

Rule Definition

The unary & operator shall not be overloaded.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-3-4

Evaluation of the operand to the sizeof operator shall not contain side effects.

Description

Rule Definition

Evaluation of the operand to the sizeof operator shall not contain side effects.

Polyspace Implementation

The checker does not show a warning on volatile accesses and function calls

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M5-8-1

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

Description

Rule Definition

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Expressions

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-2-1

Assignment operators shall not be used in sub-expressions.

Description

Rule Definition

Assignment operators shall not be used in sub-expressions.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-2-2

Floating-point expressions shall not be directly or indirectly tested for equality or inequality.

Description

Rule Definition

Floating-point expressions shall not be directly or indirectly tested for equality or inequality.

Polyspace Implementation

The checker detects the use of == or != with floating-point variables or expressions. The checker does not detect indirectly testing of equality, for instance, using the <= operator.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-2-3

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.

Description

Rule Definition

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.

Polyspace Implementation

The checker considers a null statement as a line where the first character excluding comments is a semicolon. The checker flags situations where:

- Comments appear before the semicolon.

  For instance:

  ```
  /* wait for pin */ ;
  ```

- Comments appear immediately after the semicolon without a white space in between.

  For instance:

  ```
  ;// wait for pin
  ```

The checker also shows a violation when a second statement appears on the same line following the null statement.

For instance:

```
; count++;
```

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-3-1

The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Description

Rule Definition

The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Rationale

A compound statement is included in braces.

If a block of code associated with an iteration or selection statement is not contained in braces, you can make mistakes about the association. For example:

- You can wrongly associate a line of code with an iteration or selection statement because of its indentation.
- You can accidentally place a semicolon following the iteration or selection statement. Because of the semicolon, the line following the statement is no longer associated with the statement even though you intended otherwise.

This checker enforces the practice of adding braces following a selection or iteration statement even for a single line in the body. Later, when more lines are added, the developer adding them does not need to note the absence of braces and include them.

Polyspace Implementation

The checker flags `for` loops where the first token following a `for` statement is not a left brace, for instance:

```
for (i=init_val; i > 0; i--)
   if (arr[i] < 0)
      arr[i] = 0;
```

Similar checks are performed for `switch`, `for` and `do..while` statements.

The second line of the message on the **Result Details** pane indicates which statement is violating the rule. For instance, in the preceding example, the second line of the message states that the `for` loop is violating the rule.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-4-1

An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

Description

Rule Definition

An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-4-2

All if ... else if constructs shall be terminated with an else clause.

Description

Rule Definition

All if ... else if constructs shall be terminated with an else clause.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-4-3

A switch statement shall be a well-formed switch statement.

Description

Rule Definition

A switch statement shall be a well-formed switch statement.

Polyspace Implementation

The checker flags these situations:

- A statement occurs between the `switch` statement and the first `case` statement.

  For instance:

  ```
  switch(ch) {
    int temp;
    case 1:
      break;
    default:
      break;
  }
  ```
- A label or a jump statement such as `goto` or `return` occurs in the `switch` block.
- A variable is declared in a `case` statement (outside any block).

  For instance:

  ```
  switch(ch) {
    case 1:
      int temp;
      break;
    default:
      break;
  }
  ```

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-4-4

A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

Description

Rule Definition

A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-4-5

An unconditional throw or break statement shall terminate every non-empty switch-clause.

Description

Rule Definition

An unconditional throw or break statement shall terminate every non-empty switch-clause.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-4-6

The final clause of a switch statement shall be the default-clause.

Description

Rule Definition

The final clause of a switch statement shall be the default-clause.

Polyspace Implementation

The checker detects `switch` statements that do not have a final `default` clause.

The checker does not raise a violation if the `switch` variable is an `enum` with finite number of values and you have a `case` clause for each value. For instance:

```
enum Colours { RED, BLUE, GREEN } colour;

switch ( colour ) {
      case RED:
        break;
      case BLUE:
        break;
      case GREEN:
        break;
}
```

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-4-7

The condition of a switch statement shall not have bool type.

Description

Rule Definition

The condition of a switch statement shall not have bool type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-5-2

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.

Description

Rule Definition

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-5-3

The loop-counter shall not be modified within condition or statement.

Description

Rule Definition

The loop-counter shall not be modified within condition or statement.

Rationale

The `for` loop has a specific syntax for modifying the loop counter. A code reviewer expects modification using that syntax. Modifying the loop counter elsewhere can make the code harder to review.

Polyspace Implementation

The checker flags modification of a `for` loop counter in the loop body or the loop condition (the condition that is checked to see if the loop must be terminated).

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-5-4

The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop.

Description

Rule Definition

The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-5-5

A loop-control-variable other than the loop-counter shall not be modified within condition or expression.

Description

Rule Definition

A loop-control-variable other than the loop-counter shall not be modified within condition or expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-5-6

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

Description

Rule Definition

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-6-1

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

Description

Rule Definition

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-6-2

The goto statement shall jump to a label declared later in the same function body.

Description

Rule Definition

The goto statement shall jump to a label declared later in the same function body.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M6-6-3

The continue statement shall only be used within a well-formed for loop.

Description

Rule Definition

The continue statement shall only be used within a well-formed for loop.

Polyspace Implementation

The checker flags the use of `continue` statements in:

- `for` loops that are not well-formed, that is, loops that violate rules 6-5-x.
- `while` loops.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Statements

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M7-1-2

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Description

Rule Definition

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Polyspace Implementation

The checker flags pointers where the underlying object is not const-qualified but never modified in the function body.

If a variable is passed to another function by reference or pointers, the checker assumes that the variable can be modified. Pointers that point to these variables are not flagged.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M7-3-1

The global namespace shall only contain main, namespace declarations and extern "C" declarations.

Description

Rule Definition

The global namespace shall only contain main, namespace declarations and extern "C" declarations.

Rationale

The rule makes sure that all names found at global scope are part of a namespace. Adhering to this rule avoids name clashes and ensures that developers do not reuse a variable name, resulting in compilation/linking errors, or shadow a variable name, resulting in possibly unexpected issues later.

Polyspace Implementation

Other than the `main` function, the checker flags all names used at global scope that are not part of a namespace.

The checker does not flag names at global scope if they are declared in `extern "C"` blocks (C code included within C++ code). However, if you use the option `Ignore link errors (-no-extern-c)`, these names are also flagged.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M7-3-2

The identifier main shall not be used for a function other than the global function main.

Description

Rule Definition

The identifier main shall not be used for a function other than the global function main.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M7-3-3

There shall be no unnamed namespaces in header files.

Description

Rule Definition

There shall be no unnamed namespaces in header files.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M7-3-4

Using-directives shall not be used.

Description

Rule Definition

using-directives shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M7-3-6

Using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.

Description

Rule Definition

using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M7-4-2

Assembler instructions shall only be introduced using the asm declaration.

Description

Rule Definition

Assembler instructions shall only be introduced using the asm declaration.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M7-4-3

Assembly language shall be encapsulated and isolated.

Description

Rule Definition

Assembly language shall be encapsulated and isolated.

Polyspace Implementation

The checker flags `asm` statements unless they are encapsulated in a function call.

For instance, the noncompliant `asm` statement below is in regular C code while the compliant `asm` statement is encapsulated in a call to the function `Delay`.

```
void Delay ( void )
  {
    asm( "NOP");//Compliant
  }
void fn (void)
  {
    DoSomething();
    Delay();// Assembler is encapsulated
    DoSomething();
    asm("NOP"); //Noncompliant
    DoSomething();
  }
```

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M7-5-1

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Description

Rule Definition

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declaration

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M8-0-1

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

Description

Rule Definition

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M8-3-1

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

Description

Rule Definition

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M8-4-2

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.

Description

Rule Definition

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.

Polyspace Implementation

The checker detects mismatch in parameter names between:

- A function declaration and the corresponding definition.
- Two declarations of a function, provided they occur in the same file.

  If the declarations occur in different files, the checker does not raise a violation for mismatch in parameter names. Redeclarations in different files are forbidden by `AUTOSAR C++14 Rule M3-2-3`.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M8-4-4

A function identifier shall either be used to call the function or it shall be preceded by &.

Description

Rule Definition

A function identifier shall either be used to call the function or it shall be preceded by &.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M8-5-2

Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

Description

Rule Definition

Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Declarators

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M9-3-1

Const member functions shall not return non-const pointers or references to class-data.

Description

Rule Definition

const member functions shall not return non-const pointers or references to class-data.

Polyspace Implementation

The checker flags a rule violation only if a `const` member function returns a non-`const` pointer or reference to a nonstatic data member. The rule does not apply to static data members.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Classes

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

AUTOSAR C++14 Rule M9-3-3

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.

Description

Rule Definition

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.

Polyspace Implementation

The checker flags member functions that are not declared static but do not access a data member of the class. Such a function can be potentially declared static.

The checker flags member functions that are not declared const but do not modify a data member of the class. Such a function can be potentially declared const.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Standard Violations Not Displayed”.

Check Information

Group: Classes

See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961

Acknowledgment

Extracts from the standard "ISO/IEC TS 17961 Technical Specification - 2013-11-15" are reproduced with the agreement of AFNOR. Only the original and complete text of the standard, as published by AFNOR Editions - accessible via the website www.boutique.afnor.org - has normative value.

ISO/IEC TS 17961 [accfree]

Accessing freed memory

Description

Rule Definition

Accessing freed memory.

Polyspace Implementation

This checker checks for these issues:

- **Use of previously freed pointer**.
- **Invalid use of standard library string routine**.

Examples

Use of previously freed pointer

Issue

Use of previously freed pointer occurs when you access a block of memory after freeing the block using the `free` function.

Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to access this block of memory can result in unpredictable behavior or even a segmentation fault.

Fix

The fix depends on the root cause of the defect. See if you intended to free the memory later or allocate another memory block to the pointer before access.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before dereferencing pointers, check them for NULL values and handle the error. In this way, you are protected against accessing a freed block.

Example - Use of Previously Freed Pointer Error

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   {
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);
```

```
    j = *pi + shift;
    /* Defect: Reading a freed pointer */

    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, dereferencing`pi` after the `free` statement is not valid.

Correction — Free Pointer After Use

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
#include <stdlib.h>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

Invalid use of standard library string routine

Issue

Invalid use of standard library string routine occurs when a string library function is called with invalid arguments.

Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Invalid Use of Standard Library String Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_String(void)
{
 char *res;
 char gbuffer[5],text[20]="ABCDEFGHIJKL";

 res=strcpy(gbuffer,text);
 /* Error: Size of text is less than gbuffer */

 return(res);
}
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
#include <stdio.h>

char* Copy_String(void)
{
 char *res;
 /*Fix: gbuffer has equal or larger size than text */
 char gbuffer[20],text[20]="ABCDEFGHIJKL";

 res=strcpy(gbuffer,text);

 return(res);
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

"Check for Coding Standard Violations"

Introduced in R2019a

ISO/IEC TS 17961 [accsig]

Accessing shared objects in signal handlers

Description

Rule Definition

Accessing shared objects in signal handlers.

Polyspace Implementation

This checker checks for **Shared data access within signal handler**.

Examples

Shared data access within signal handler

Issue

Shared data access within signal handler occurs when you access or modify a shared object inside a signal handler.

Risk

When you define a signal handler function to access or modify a shared object, the handler accesses or modifies the shared object when it receives a signal. If another function is already accessing the shared object, that function causes a race condition and can leave the data in an inconsistent state.

Fix

To access or modify shared objects inside a signal handler, check that the objects are lock-free atomic, or, if they are integers, declare them as `volatile sig_atomic_t`.

Example - `int` Variable Access in Signal Handler

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* declare global variable. */
int e_flag;

void sig_handler(int signum)
{
    /* Signal handler accesses variable that is not
    of type volatile sig_atomic_t. */
    e_flag = signum;
}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
```

```
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

In this example, `sig_handler` accesses `e_flag`, a variable of type `int`. A concurrent access by another function can leave `e_flag` in an inconsistent state.

Correction — Declare Variable of Type `volatile sig_atomic_t`

Before you access a shared variable from a signal handler, declare the variable with type `volatile sig_atomic_t` instead of `int`. You can safely access variables of this type asynchronously.

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* Declare variable of type volatile sig_atomic_t. */
volatile sig_atomic_t e_flag;
void sig_handler(int signum)
{
    /* Use variable of proper type inside signal handler. */
    e_flag = signum;

}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [addrescape]

Escaping of the address of an automatic object

Description

Rule Definition

Escaping of the address of an automatic object.

Polyspace Implementation

This checker checks for these issues:

- **Pointer or reference to stack variable leaving scope**.
- **Use of automatic variable as putenv-family function argument**.

Examples

Pointer or reference to stack variable leaving scope

Issue

Pointer or reference to stack variable leaving scope occurs when a pointer or reference to a local variable leaves the scope of the variable. For instance:

- A function returns a pointer to a local variable.
- A function performs the assignment `globPtr = &locVar`. `globPtr` is a global pointer variable and `locVar` is a local variable.
- A function performs the assignment `*paramPtr = &locVar`. `paramPtr` is a function parameter that is, for instance, an `int**` pointer and `locVar` is a local `int` variable.
- A C++ method performs the assignment `memPtr = &locVar`. `memPtr` is a pointer data member of the class the method belongs to. `locVar` is a variable local to the method.

The defect also applies to memory allocated using the `alloca` function. The defect does not apply to static, local variables.

Risk

Local variables are allocated an address on the stack. Once the scope of a local variable ends, this address is available for reuse. Using this address to access the local variable value outside the variable scope can cause unexpected behavior.

If a pointer to a local variable leaves the scope of the variable, Polyspace Bug Finder highlights the defect. The defect appears even if you do not use the address stored in the pointer. For maintainable code, it is a good practice to not allow the pointer to leave the variable scope. Even if you do not use the address in the pointer now, someone else using your function can use the address, causing undefined behavior.

Fix

Do not allow a pointer or reference to a local variable to leave the variable scope.

Example - Pointer to Local Variable Returned from Function

```
void func2(int *ptr) {
    *ptr = 0;
}

int* func1(void) {
    int ret = 0;
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

In this example, `func1` returns a pointer to local variable `ret`.

In `main`, `ptr` points to the address of the local variable. When `ptr` is accessed in `func2`, the access is illegal because the scope of `ret` is limited to `func1`,

Use of automatic variable as putenv-family function argument

Issue

Use of automatic variable as `putenv`-family function argument occurs when the argument of a `putenv`-family function is a local variable with automatic duration.

Risk

The function `putenv(char *string)` inserts a pointer to its supplied argument into the environment array, instead of making a copy of the argument. If the argument is an automatic variable, its memory can be overwritten after the function containing the `putenv()` call returns. A subsequent call to `getenv()` from another function returns the address of an out-of-scope variable that cannot be dereferenced legally. This out-of-scope variable can cause environment variables to take on unexpected values, cause the program to stop responding, or allow arbitrary code execution vulnerabilities.

Fix

Use `setenv()`/`unsetenv()` to set and unset environment variables. Alternatively, use `putenv`-family function arguments with dynamically allocated memory, or, if your application has no reentrancy requirements, arguments with static duration. For example, a single thread execution with no recursion or interrupts does not require reentrancy. It cannot be called (reentered) during its execution.

Example - Automatic Variable as Argument of `putenv()`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024

void func(int var)
{
    char env[SIZE1024];
    int retval = sprintf(env, "TEST=%s", var ? "1" : "0");
    if (retval <= 0) {
```

```
        /* Handle error */
    }
    /* Environment variable TEST is set using putenv().
    The argument passed to putenv is an automatic variable. */
    retval = putenv(env);
    if (retval != 0) {
        /* Handle error */
    }
}
```

In this example, `sprintf()` stores the character string `TEST=var` in `env`. The value of the environment variable `TEST` is then set to `var` by using `putenv()`. Because `env` is an automatic variable, the value of `TEST` can change once `func()` returns.

Correction — Use `static` Variable for Argument of `putenv()`

Declare `env` as a static-duration variable. The memory location of `env` is not overwritten for the duration of the program, even after `func()` returns.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024
void func(int var)
{
    /* static duration variable */
    static char env[SIZE1024];
    int retval = sprintf(env,"TEST=%s", var ? "1" : "0");
    if (retval <= 0) {
        /* Handle error */
    }

    /* Environment variable TEST is set using putenv() */
    retval=putenv(env);
    if (retval != 0) {
        /* Handle error */
    }
}
```

Correction — Use `setenv()` to Set Environment Variable Value

To set the value of `TEST` to `var`, use `setenv()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024

void func(int var)
{
    /* Environment variable TEST is set using setenv() */
    int retval = setenv("TEST", var ? "1" : "0", 1);

    if (retval != 0) {
        /* Handle error */
```

```
    }
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [alignconv]

Converting pointer values to more strictly aligned pointer types

Description

Rule Definition

Converting pointer values to more strictly aligned pointer types.

Polyspace Implementation

This checker checks for **Wrong allocated object size for cast**.

Examples

Wrong allocated object size for cast

Issue

Wrong allocated object size for cast occurs during pointer conversion when the pointer's address is misaligned. If a pointer is converted to a different pointer type, the size of the allocated memory must be a multiple of the size of the destination pointer.

Risk

Dereferencing a misaligned pointer has undefined behavior and can cause your program to crash.

Fix

Suppose you convert a pointer `ptr1` to `ptr2`. If `ptr1` points to a buffer of `N` bytes and `ptr2` is a *`type`* * pointer where `sizeof(`*`type`*`)` is `n` bytes, make sure that `N` is an integer multiple of `n`.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Dynamic Allocation of Pointers

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(13);
    long *dest;

    dest = (long*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to a `long`*. The dynamically allocated memory of `ptr`, 13 bytes, is not a multiple of the size of `dest`, 4 bytes. This misalignment causes the **Wrong allocated object size for cast** defect.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the allocated memory to 12 instead of 13.

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(12);
    long *dest;

    dest = (long*)ptr;
}
```

Example - Static Allocation of Pointers

```
void static_non_align(void){
    char arr[13], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to an `int*` in line 6. `ptr` has a memory size of 13 bytes because the array `arr` has a size of 13 bytes. The size of `dest` is 4 bytes, which is not a multiple of 13. This misalignment causes the **Wrong allocated object size for cast** defect.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the size of the array `arr` to a multiple of 4.

```
void static_non_align(void){
    char arr[12], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr;
}
```

Example - Allocation with a Function

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(13);  //defect
    dest2 = (char*)my_alloc(13); //not a defect
}
```

In this example, the software raises a defect on the conversion of the pointer returned by `my_alloc(13)` to an `int*` in line 11. `my_alloc(13)` returns a pointer with a dynamically allocated size of 13 bytes. The size of `dest1` is 4 bytes, which is not a divisor of 13. This misalignment causes the **Wrong allocated object size for cast** defect. In line 12, the same function call, `my_alloc(13)`, does not call a defect for the conversion to `dest2` because the size of `char*`, 1 byte, a divisor of 13.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the argument for `my_alloc` to a multiple of 4.

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(12);
    dest2 = (char*)my_alloc(13);
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [argcomp]

Calling functions with incorrect arguments

Description

Rule Definition

Calling functions with incorrect arguments.

Polyspace Implementation

This checker checks for these issues:

- **Conflicting declarations or conflicting declaration and definition**.
- **Unreliable cast of function pointer**.

Examples

Conflicting declarations or conflicting declaration and definition

Issue

The issue occurs when all declarations of an object or function do not use the same names and type qualifiers.

The rule checker detects situations where parameter names or data types are different between multiple declarations or the declaration and the definition. The checker considers declarations in all translation units and flags issues that are not likely to be detected by a compiler.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Risk

Consistently using parameter names and types across declarations of the same object or function encourages stronger typing. It is easier to check that the same function interface is used across all declarations.

Example - Mismatch in Parameter Names

```
extern int div (int num, int den);

int div(int den, int num) { /* Non compliant */
    return(num/den);
}
```

In this example, the rule is violated because the parameter names in the declaration and definition are switched.

Example - Mismatch in Parameter Data Types

```
typedef unsigned short width;
typedef unsigned short height;
```

```
typedef unsigned int area;

extern area calculate(width w, height h);

area calculate(width w, width h) { /* Non compliant *
    return w*h;
}
```

In this example, the rule is violated because the second argument of the `calculate` function has data type:

- `height` in the declaration.
- `width` in the definition.

The rule is violated even though the underlying type of `height` and `width` are identical.

Unreliable cast of function pointer

Issue

Unreliable cast of function pointer occurs when a function pointer is cast to another function pointer that has different argument or return type.

This defect applies only if the code language for the project is C.

Risk

If you cast a function pointer to another function pointer with different argument or return type and then use the latter function pointer to call a function, the behavior is undefined.

Fix

Avoid a cast between two function pointers with mismatch in argument or return types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Unreliable cast of function pointer error

```
#include <stdio.h>
#include <math.h>
#include <stdio.h>
#define PI 3.142

double Calculate_Sum(int (*fptr)(double))
{
    double  sum = 0.0;
    double  y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}
```

```
int main(void)
{
    double  (*fp)(double);
    double  sum;

    fp = sin;
    sum = Calculate_Sum(fp);
    /* Defect: fp implicitly cast to int(*) (double) */

    printf("sum(sin): %f\n", sum);
    return 0;
}
```

The function pointer `fp` is declared as `double (*)(double)`. However in passing it to function `Calculate_Sum`, `fp` is implicitly cast to `int (*)(double)`.

Correction — Avoid Function Pointer Cast

One possible correction is to check that the function pointer in the definition of `Calculate_Sum` has the same argument and return type as `fp`. This step makes sure that `fp` is not implicitly cast to a different argument or return type.

```
#include <stdio.h>
#include <math.h>
#include <stdio.h>
# define PI 3.142

/*Fix: fptr has same argument and return type everywhere*/
double Calculate_Sum(double (*fptr)(double))
{
    double  sum = 0.0;
    double y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;


    fp = sin;
    sum = Calculate_Sum(fp);
    printf("sum(sin): %f\n", sum);

    return 0;
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [asyncsig]

Calling functions in the C Standard Library other than abort, _Exit, and signal from within a signal handler

Description

Rule Definition

Calling functions in the C Standard Library other than abort, _Exit, and signal from within a signal handler.

Polyspace Implementation

This checker checks for these issues:

- **Function called from signal handler not asynchronous-safe (strict)**.
- **Function called from signal handler not asynchronous-safe**.

Examples

Function called from signal handler not asynchronous-safe (strict)

Issue

Function called from signal handler not asynchronous-safe (strict) occurs when a signal handler calls a function that is not asynchronous-safe according to the C standard. An asynchronous-safe function can be interrupted at any point in its execution, then called again without causing an inconsistent state. It can also correctly handle global data that might be in an inconsistent state.

When you select the checker **Function called from signal handler not asynchronous-safe**, the checker detects calls to functions that are not asynchronous-safe according to the POSIX standard. **Function called from signal handler not asynchronous-safe (strict)** does not raise a defect for these cases. **Function called from signal handler not asynchronous-safe (strict)** raises a defect for functions that are asynchronous-safe according to the POSIX standard but not according to the C standard.

If a signal handler calls another function that calls an asynchronous-unsafe function, the defect appears on the function call in the signal handler. The defect traceback shows the full path from the signal handler to the asynchronous-unsafe function.

Risk

When a signal handler is invoked, the execution of the program is interrupted. After the handler is finished, program execution resumes at the point of interruption. If a function is executing at the time of the interruption, calling it from within the signal handler is undefined behavior, unless it is asynchronous-safe.

Fix

The C standard defines the following functions as asynchronous-safe. You can call these functions from a signal handler:

- abort()
- _Exit()
- quick_exit()
- signal()

Example - Call to raise() Inside Signal Handler

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

void SIG_ERR_handler(int signum)
{
    int s0 = signum;
    /* SIGTERM specific handling */
}

void sig_handler(int signum)
{
    int s0 = signum;
    /* Call raise() */
    if (raise(SIGTERM) != 0) {
        /* Handle error */
    }
}

int finc(void)
{
    if (signal(SIGTERM, SIG_ERR_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return 0;
}
```

In this example, sig_handler calls raise() when catching a signal. If the handler catches another signal while raise() is executing, the behavior of the program is undefined.

Correction — Remove Call to `raise()` in Signal Handler

According to the C standard, the only functions that you can safely call from a signal handler are `abort()`, `_Exit()`, `quick_exit()`, and `signal()`.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

void SIG_ERR_handler(int signum)
{
    int s0 = signum;
    /* SIGTERM specific handling */
}
void sig_handler(int signum)
{
    int s0 = signum;


}

int func(void)
{
    if (signal(SIGTERM, SIG_ERR_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return 0;
}
```

Function called from signal handler not asynchronous-safe

Issue

Function called from signal handler not asynchronous-safe occurs when a signal handler calls a function that is not asynchronous-safe according to the POSIX standard. An asynchronous-safe function can be interrupted at any point in its execution, then called again without causing an inconsistent state. It can also correctly handle global data that might be in an inconsistent state.

If a signal handler calls another function that calls an asynchronous-unsafe function, the defect appears on the function call in the signal handler. The defect traceback shows the full path from the signal handler to the asynchronous-unsafe function.

Risk

When a signal handler is invoked, the execution of the program is interrupted. After the handler is finished, program execution resumes at the point of interruption. If a function is executing at the time of the interruption, calling it from within the signal handler is undefined behavior, unless it is asynchronous-safe.

Fix

The POSIX standard defines these functions as asynchronous-safe. You can call these functions from a signal handler.

_exit()	getpgrp()	setsockopt()
_Exit()	getpid()	setuid()
abort()	getppid()	shutdown()
accept()	getsockname()	sigaction()
access()	getsockopt()	sigaddset()
aio_error()	getuid()	sigdelset()
aio_return()	kill()	sigemptyset()
aio_suspend()	link()	sigfillset()
alarm()	linkat()	sigismember()
bind()	listen()	signal()
cfgetispeed()	lseek()	sigpause()
cfgetospeed()	lstat()	sigpending()
cfsetispeed()	mkdir()	sigprocmask()
cfsetospeed()	mkdirat()	sigqueue()
chdir()	mkfifo()	sigset()
chmod()	mkfifoat()	sigsuspend()
chown()	mknod()	sleep()
clock_gettime()	mknodat()	sockatmark()
close()	open()	socket()
connect()	openat()	socketpair()
creat()	pathconf()	stat()
dup()	pause()	symlink()
dup2()	pipe()	symlinkat()
execl()	poll()	sysconf()
execle()	posix_trace_event()	tcdrain()
execv()	pselect()	tcflow()
execve()	pthread_kill()	tcflush()
faccessat()	pthread_self()	tcgetattr()
fchdir()	pthread_sigmask()	tcgetpgrp()
fchmod()	quick_exit()	tcsendbreak()

fchmodat()	raise()	tcsetattr()
fchown()	read()	tcsetpgrp()
fchownat()	readlink()	time()
fcntl()	readlinkat()	timer_getoverrun()
fdatasync()	recv()	timer_gettime()
fexecve()	recvfrom()	timer_settime()
fork()	recvmsg()	times()
fpathconf()	rename()	umask()
fstat()	renameat()	uname()
fstatat()	rmdir()	unlink()
fsync()	select()	unlinkat()
ftruncate()	sem_post()	utime()
futimens()	send()	utimensat()
getegid()	sendmsg()	utimes()
geteuid()	sendto()	wait()
getgid()	setgid()	waitpid()
getgroups()	setpgid()	write()
getpeername()	setsid()	

Functions not in the previous table are not asynchronous-safe, and should not be called from a signal hander.

Example - Call to `printf()` Inside Signal Handler

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

#define SIZE20 20

extern volatile sig_atomic_t e_flag;

void display_info(const char *info)
{
    if (info)
    {
        (void)fputs(info, stderr);
    }
}

void sig_handler(int signum)
{
    /* Call function printf() that is not
    asynchronous-safe */
    printf("signal %d received.", signum);
```

```
    e_flag = 1;
}

int main(void)
{
    e_flag = 0;
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info = (char *)calloc(SIZE20, sizeof(char));
    if (info == NULL)
    {
        /* Handle Error */
    }
    while (!e_flag)
    {
        /* Main loop program code */
        display_info(info);
        /* More program code */
    }
    free(info);
    info = NULL;
    return 0;
}
```

In this example, `sig_handler` calls `printf()` when catching a signal. If the handler catches another signal while `printf()` is executing, the behavior of the program is undefined.

Correction — Set Flag Only in Signal Handler

Use your signal handler to set only the value of a flag. `e_flag` is of type volatile `sig_atomic_t`. `sig_handler` can safely access it asynchronously.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

#define SIZE20 20

extern volatile sig_atomic_t e_flag;

void display_info(const char *info)
{
    if (info)
    {
        (void)fputs(info, stderr);
    }
}

void sig_handler1(int signum)
{
```

```
    int s0 = signum;
    e_flag = 1;
}

int func(void)
{
    e_flag = 0;
    if (signal(SIGINT, sig_handler1) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info = (char *)calloc(SIZE20, 1);
    if (info == NULL)
    {
        /* Handle error */
    }
    while (!e_flag)
    {
        /* Main loop program code */
        display_info(info);
        /* More program code */
    }
    free(info);
    info = NULL;
    return 0;
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [boolasgn]

No assignment in conditional expressions

Description

Rule Definition

No assignment in conditional expressions.

Polyspace Implementation

This checker checks for **Invalid use of = (assignment) operator**.

Examples

Invalid use of = (assignment) operator

Issue

Invalid use of = operator occurs when an assignment is made inside the predicate of a conditional, such as `if` or `while`.

In C and C++, a single equal sign is an assignment not a comparison. Using a single equal sign in a conditional statement can indicate a typo or a mistake.

Risk

- Conditional statement tests the wrong values— The single equal sign operation assigns the value of the right operand to the left operand. Then, because this assignment is inside the predicate of a conditional, the program checks whether the new value of the left operand is nonzero or not NULL.
- Maintenance and readability issues — Even if the assignment is intended, someone reading or updating the code can misinterpret the assignment as an equality comparison instead of an assignment.

Fix

- If the assignment is a bug, to check for equality, add a second equal sign (==).
- If the assignment inside the conditional statement was intentional, to improve readability, separate the assignment and the test. Move the assignment outside the control statement. In the control statement, simply test the result of the assignment.

  If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Single Equal Sign Inside an `if` Condition

```
#include <stdio.h>

void bad_equals_ex(int alpha, int beta)
{
    if(alpha = beta)
```

```
    {
        printf("Equal\n");
    }
}
```

The equal sign is flagged as a defect because the assignment operator is used within the predicate of the if-statement. The predicate assigns the value `beta` to `alpha`, then implicitly tests whether `alpha` is true or false.

Correction — Change Expression to Comparison

One possible correction is adding an additional equal sign. This correction changes the assignment to a comparison. The if condition compares whether `alpha` and `beta` are equal.

```
#include <stdio.h>

void equality_test(int alpha, int beta)
{
    if(alpha == beta)
    {
        printf("Equal\n");
    }
}
```

Correction — Assignment and Comparison Inside the `if` Condition

If an assignment must be made inside the predicate, a possible correction is adding an explicit comparison. This correction assigns the value of `beta` to `alpha`, then explicitly checks whether `alpha` is nonzero. The code is clearer.

```
#include <stdio.h>

int assignment_not_zero(int alpha, int beta)
{
    if((alpha = beta) != 0)
    {
        return alpha;
    }
    else
    {
        return 0;
    }
}
```

Correction — Move Assignment Outside the `if` Statement

If the assignment can be made outside the control statement, one possible correction is to separate the assignment and comparison. This correction assigns the value of `beta` to `alpha` before the if. Inside the if-condition, only `alpha` is given to test if `alpha` is nonzero or not NULL.

```
#include <stdio.h>

void assign_and_print(int alpha, int beta)
{
    alpha = beta;
    if(alpha)
    {
        printf("%d", alpha);
```

```
    }
}
```

Check Information

Decidability: Decidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [chreof]

Using character values that are indistinguishable from EOF

Description

Rule Definition

Using character values that are indistinguishable from EOF.

Polyspace Implementation

This checker checks for **Character value absorbed into EOF**.

Examples

Character value absorbed into EOF

Issue

Character value absorbed into EOF occurs when you perform a data type conversion that makes a valid character value indistinguishable from `EOF` (End-of-File). Bug Finder flags the defect in one of the following situations:

- *End-of-File*: You perform a data type conversion such as from `int` to `char` that converts a non-EOF character value into `EOF`.

  ```
  char ch = (char)getchar()
  ```

  You then compare the result with EOF.

  ```
  if((int)ch == EOF)
  ```

  The conversion can be explicit or implicit.
- *Wide End-of-File*: You perform a data type conversion that can convert a non-WEOF wide character value into WEOF, and then compare the result with WEOF.

Risk

The data type `char` cannot hold the value `EOF` that indicates the end of a file. Functions such as `getchar` have return type `int` to accommodate `EOF`. If you convert from `int` to `char`, the values `UCHAR_MAX` (a valid character value) and `EOF` get converted to the same value -1 and become indistinguishable from each other. When you compare the result of this conversion with `EOF`, the comparison can lead to false detection of `EOF`. This rationale also applies to wide character values and `WEOF`.

Fix

Perform the comparison with `EOF` or `WEOF` before conversion.

Example - Return Value of `getchar` Converted to `char`

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define fatal_error() abort()

char func(void)
{
    char ch;
    ch = getchar();
    if (EOF == (int)ch) {
        fatal_error();
    }
    return ch;
}
```

In this example, the return value of `getchar` is implicitly converted to `char`. If `getchar` returns `UCHAR_MAX`, it is converted to -1, which is indistinguishable from EOF. When you compare with EOF later, it can lead to a false positive.

Correction — Perform Comparison with EOF Before Conversion

One possible correction is to first perform the comparison with EOF, and then convert from `int` to `char`.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

char func(void)
{
    int i;
    i = getchar();
    if (EOF == i) {
        fatal_error();
    }
    else {
        return (char)i;
    }
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [chrsgnext]

Passing arguments to character handling functions that are not representable as unsigned char

Description

Rule Definition

Passing arguments to character handling functions that are not representable as unsigned char.

Polyspace Implementation

This checker checks for **Invalid use of standard library integer routine**.

Examples

Invalid use of standard library integer routine

Issue

Invalid use of standard library integer routine occurs when you use invalid arguments with an integer function from the standard library. This defect picks up:

- Character Conversion

  ```
  toupper, tolower
  ```
- Character Checks

  ```
  isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct,
  isspace, isupper, isxdigit
  ```
- Integer Division

  ```
  div, ldiv
  ```
- Absolute Values

  ```
  abs, labs
  ```

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Absolute Value of Large Negative

```
#include <limits.h>
#include <stdlib.h>
```

```
int absoluteValue(void) {

    int neg = INT_MIN;
    return abs(neg);
}
```

The input value to `abs` is `INT_MIN`. The absolute value of `INT_MIN` is `INT_MAX+1`. This number cannot be represented by the type `int`.

Correction — Change Input Argument

One possible correction is to change the input value to fit returned data type. In this example, change the input value to `INT_MIN+1`.

```
#include <limits.h>
#include <stdlib.h>

int absoluteValue(void) {

    int neg = INT_MIN+1;
    return abs(neg);
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [dblfree]

Freeing memory multiple times

Description

Rule Definition

Freeing memory multiple times.

Polyspace Implementation

This checker checks for **Deallocation of previously deallocated pointer**.

Examples

Deallocation of previously deallocated pointer

Issue

Deallocation of previously deallocated pointer occurs when a block of memory is freed more than once using the `free` function without an intermediate allocation.

Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to free this block of memory can result in a segmentation fault.

Fix

The fix depends on the root cause of the defect. See if you intended to allocate a memory block to the pointer between the first deallocation and the second. Otherwise, remove the second `free` statement.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before freeing pointers, check them for NULL values and handle the error. In this way, you are protected against freeing an already freed block.

Example - Deallocation of Previously Deallocated Pointer Error

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    free (pi);
    /* Defect: pi has already been freed */
}
```

The first `free` statement releases the block of memory that `pi` refers to. The second `free` statement on `pi` releases a block of memory that has been freed already.

Correction — Remove Duplicate Deallocation

One possible correction is to remove the second `free` statement.

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    /* Fix: remove second deallocation */
 }
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [diverr]

Integer division errors

Description

Rule Definition

Integer division errors.

Polyspace Implementation

This checker checks for **Integer division by zero**.

Examples

Integer division by zero

Issue

Integer division by zero occurs when the denominator of a division or modulo operation can be a zero-valued integer.

Risk

A division by zero can result in a program crash.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the denominator variable acquires a zero value. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

It is a good practice to check for zero values of a denominator before division and handle the error. Instead of performing the division directly:

```
res = num/den;
```

use a library function that handles zero values of the denominator before performing the division:

```
res = div(num, den);
```

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Dividing an Integer by Zero

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;
```

```
    result = num/denom;

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

Correction — Check Before Division

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    if (denom != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
int fraction(int num)
{
    int denom = 2;
    int result = 0;

    result = num/denom;

    return result;
}
```

Example - Modulo Operation with Zero

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        arr[i] = input % i;
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

In this example, Polyspace flags the modulo operation as a division by zero. Because modulo is inherently a division operation, the divisor (right hand argument) cannot be zero. The modulo operation uses the `for` loop index as the divisor. However, the `for` loop starts at zero, which cannot be an iterator.

Correction — Check Divisor Before Operation

One possible correction is checking the divisor before the modulo operation. In this example, see if the index `i` is zero before the modulo operation.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        if(i != 0)
        {
             arr[i] = input % i;
        }
        else
        {
             arr[i] = input;
        }
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

Correction — Change Divisor

Another possible correction is changing the divisor to a nonzero integer. In this example, add one to the index before the `%` operation to avoid dividing by zero.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
         arr[i] = input % (i+1);
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [fileclose]

Failing to close files or free dynamic memory when they are no longer needed

Description

Rule Definition

Failing to close files or free dynamic memory when they are no longer needed.

Polyspace Implementation

This checker checks for these issues:

- **Memory leak**.
- **Resource leak**.
- **Thread-specific memory leak**.

Examples

Memory leak

Issue

Memory leak occurs when you do not free a block of memory allocated through `malloc`, `calloc`, `realloc`, or `new`. If the memory is allocated in a function, the defect does not occur if:

- Within the function, you free the memory using `free` or `delete`.
- The function returns the pointer assigned by `malloc`, `calloc`, `realloc`, or `new`.
- The function stores the pointer in a global variable or in a parameter.

Risk

Dynamic memory allocation functions such as `malloc` allocate memory on the heap. If you do not release the memory after use, you reduce the amount of memory available for another allocation. On embedded systems with limited memory, you might end up exhausting available heap memory even during program execution.

Fix

Determine the scope where the dynamically allocated memory is accessed. Free the memory block at the end of this scope.

To free a block of memory, use the `free` function on the pointer that was used during memory allocation. For instance:

```
ptr = (int*)malloc(sizeof(int));
...
free(ptr);
```

It is a good practice to allocate and free memory in the same module at the same level of abstraction. For instance, in this example, `func` allocates and frees memory at the same level but `func2` does not.

```
void func() {
  ptr = (int*)malloc(sizeof(int));
  {
    ...
  }
  free(ptr);
}

void func2() {
  {
   ptr = (int*)malloc(sizeof(int));
   ...
  }
  free(ptr);
}
```

See CERT-C Rule MEM00-C.

Example - Dynamic Memory Not Released Before End of Function

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }


    *pi = 42;
    /* Defect: pi is not freed */
}
```

In this example, `pi` is dynamically allocated by `malloc`. The function `assign_memory` does not free the memory, nor does it return `pi`.

Correction — Free Memory

One possible correction is to free the memory referenced by `pi` using the `free` function. The `free` function must be called before the function `assign_memory` terminates

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }
    *pi = 42;

    /* Fix: Free the pointer pi*/
```

```
    free(pi);
}
```

Correction — Return Pointer from Dynamic Allocation

Another possible correction is to return the pointer `pi`. Returning `pi` allows the function calling `assign_memory` to free the memory block using `pi`.

```
#include<stdlib.h>
#include<stdio.h>

int* assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
            printf("Memory allocation failed");
            return(pi);
        }
    *pi = 42;

    /* Fix: Return the pointer pi*/
    return(pi);
}
```

Example - Memory Leak with New/Delete

```
#define NULL '\0'

void initialize_arr1(void)
{
    int *p_scalar = new int(5);
}

void initialize_arr2(void)
{
    int *p_array = new int[5];
}
```

In this example, the functions create two variables, `p_scalar` and `p_array`, using the `new` keyword. However, the functions end without cleaning up the memory for these pointers. Because the functions used `new` to create these variables, you must clean up their memory by calling `delete` at the end of each function.

Correction — Add Delete

To correct this error, add a `delete` statement for every `new` initialization. If you used brackets `[]` to instantiate a variable, you must call delete with brackets as well.

```
#define NULL '\0'

void initialize_arrs(void)
{
    int *p_scalar = new int(5);
    int *p_array = new int[5];

    delete p_scalar;
```

```
    p_scalar = NULL;

    delete[] p_array;
    p_scalar = NULL;
}
```

Resource leak

Issue

Resource leak occurs when you open a file stream by using a `FILE` pointer but do not close it before:

- The end of the pointer's scope.
- Assigning the pointer to another stream.

Risk

If you do not release file handles explicitly as soon as possible, a failure can occur due to exhaustion of resources.

Fix

Close a `FILE` pointer before the end of its scope, or before you assign the pointer to another stream.

Example - `FILE` Pointer Not Released Before End of Scope

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

In this example, the file pointer `fp1` is pointing to a file `data1.txt`. Before `fp1` is explicitly dissociated from the file stream of `data1.txt`, it is used to access another file `data2.txt`.

Correction — Release `FILE` Pointer

One possible correction is to explicitly dissociate `fp1` from the file stream of `data1.txt`.

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );
    fclose(fp1);

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

Thread-specific memory leak

Issue

Thread-specific memory leak occurs when you do not free thread-specific dynamically allocated memory before the end of a thread.

To create thread-specific storage, you generally do these steps:

1. You create a key for thread-specific storage.
2. You create the threads.
3. In each thread, you allocate storage dynamically and then associate the key with this storage.

   After the association, you can read the stored data later using the key.
4. Before the end of the thread, you free the thread-specific memory using the key.

The checker flags execution paths in the thread where the last step is missing.

The checker works on these families of functions:

- `tss_get` and `tss_set` (C11)
- `pthread_getspecific` and `pthread_setspecific` (POSIX)

Risk

The data stored in the memory is available to other processes even after the threads end (memory leak). Besides security vulnerabilities, memory leaks can shrink the amount of available memory and reduce performance.

Fix

Free dynamically allocated memory before the end of a thread.

You can explicitly free dynamically allocated memory with functions such as `free`.

Alternatively, when you create a key, you can associate a destructor function with the key. The destructor function is called with the key value as argument at the end of a thread. In the body of the destructor function, you can free any memory associated with the key. If you use this method, Bug Finder still flags a defect. Ignore this defect with appropriate comments. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Memory Not Freed at End of Thread

```
#include <threads.h>
#include <stdlib.h>

/* Global key to the thread-specific storage */
tss_t key;
enum { MAX_THREADS = 3 };


int add_data(void) {
  int *data = (int *)malloc(2 * sizeof(int));
  if (data == NULL) {
    return -1;  /* Report error  */
  }
```

```
  data[0] = 0;
  data[1] = 1;

  if (thrd_success != tss_set(key, (void *)data)) {
    /* Handle error */
  }
  return 0;
}

void print_data(void) {
  /* Get this thread's global data from key */
  int *data = tss_get(key);

  if (data != NULL) {
    /* Print data */
  }
}

int func(void *dummy) {
  if (add_data() != 0) {
    return -1;  /* Report error */
  }
  print_data();
  return 0;
}

int main(void) {
  thrd_t thread_id[MAX_THREADS];

  /* Create the key before creating the threads */
  if (thrd_success != tss_create(&key, NULL)) {
    /* Handle error */
  }

  /* Create threads that would store specific storage */
  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_create(&thread_id[i], func, NULL)) {
      /* Handle error */
    }
  }

  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_join(thread_id[i], NULL)) {
      /* Handle error */
    }
  }

  tss_delete(key);
  return 0;
}
```

In this example, the start function of each thread `func` calls two functions:

- `add_data`: This function allocates storage dynamically and associates the storage with a key using the `tss_set` function.
- `print_data`: This function reads the stored data using the `tss_get` function.

At the points where `func` returns, the dynamically allocated storage has not been freed.

Correction — Free Dynamically Allocated Memory Explicitly

One possible correction is to free dynamically allocated memory explicitly before leaving the start function of a thread. See the highlighted change in the corrected version.

In this corrected version, a defect still appears on the `return` statement in the error handling section of `func`. The defect cannot occur in practice because the error handling section is entered only if dynamic memory allocation fails. Ignore this remaining defect with appropriate comments. See "Address Polyspace Results Through Bug Fixes or Justifications".

```
#include <threads.h>
#include <stdlib.h>

/* Global key to the thread-specific storage */
tss_t key;
enum { MAX_THREADS = 3 };


int add_data(void) {
  int *data = (int *)malloc(2 * sizeof(int));
  if (data == NULL) {
    return -1;  /* Report error  */
  }
  data[0] = 0;
  data[1] = 1;

  if (thrd_success != tss_set(key, (void *)data)) {
    /* Handle error */
  }
  return 0;
}

void print_data(void) {
  /* Get this thread's global data from key */
  int *data = tss_get(key);

  if (data != NULL) {
    /* Print data */
  }
}

int func(void *dummy) {
  if (add_data() != 0) {
    return -1;  /* Report error */
  }
  print_data();
  free(tss_get(key));
  return 0;
}

int main(void) {
  thrd_t thread_id[MAX_THREADS];

  /* Create the key before creating the threads */
  if (thrd_success != tss_create(&key, NULL)) {
    /* Handle error */
```

```
  }

  /* Create threads that would store specific storage */
  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_create(&thread_id[i], func, NULL)) {
      /* Handle error */
    }
  }

  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_join(thread_id[i], NULL)) {
      /* Handle error */
    }
  }

  tss_delete(key);
  return 0;
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [filecpy]

Copying a FILE object

Description

Rule Definition

Copying a FILE object.

Polyspace Implementation

This checker checks for **Dereferencing a FILE* pointer**.

Examples

Dereferencing a FILE* pointer

Issue

The issue occurs when a pointer to a FILE object is dereferenced.

Risk

The Standard states that the address of a FILE object used to control a stream can be significant. Copying that object might not give the same behavior. This rule ensures that you cannot perform such a copy.

Directly manipulating a FILE object might be incompatible with its use as a stream designator.

Example - FILE* Pointer Dereferenced

```
#include <stdio.h>

void func(void) {
    FILE *pf1;
    FILE *pf2;
    FILE f3;

    pf2 = pf1;        /* Compliant */
    f3 = *pf2;        /* Non-compliant */
    pf2->_flags=0;    /* Non-compliant */
 }
```

In this example, the rule is violated when the FILE* pointer pf2 is dereferenced.

Check Information

Decidability: Undecidable

See Also

Check ISO/IEC TS 17961 (-iso-17961)

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [funcdecl]

Declaring the same function or object in incompatible ways

Description

Rule Definition

Declaring the same function or object in incompatible ways.

Polyspace Implementation

This checker checks for these issues:

- **Indistinguishable external identifier names**.
- **Declaration mismatch**.

Examples

Indistinguishable external identifier names

Issue

The issue occurs when external identifiers are not distinct.

Risk

External identifiers are ones declared with global scope or storage class `extern`.

Polyspace considers two names as distinct if there is a difference between their first 31 characters. If the difference between two names occurs only beyond the first 31 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 6 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`.

Example - C90: First Six Characters of Identifiers Not Unique

```
int engine_temperature_raw;
int engine_temperature_scaled;   /* Non-compliant */
int engin2_temperature;          /* Compliant */
```

In this example, the identifier `engine_temperature_scaled` has the same first six characters as a previous identifier, `engine_temperature_raw`.

Example - C99: First 31 Characters of Identifiers Not Unique

```
int engine_exhaust_gas_temperature_raw;
int engine_exhaust_gas_temperature_scaled; /* Non-compliant */

int eng_exhaust_gas_temp_raw;
int eng_exhaust_gas_temp_scaled;           /* Compliant */
```

In this example, the identifier `engine_exhaust_gas_temperature_scaled` has the same first 31 characters as a previous identifier, `engine_exhaust_gas_temperature_raw`.

Example - C90: First Six Characters Identifiers in Different Translation Units Differ in Case Alone

```
/* file1.c */
int abc = 0;

/* file2.c */
int ABC = 0; /* Non-compliant */
```

In this example, the implementation supports 6 significant case-insensitive characters in *external identifiers*. The identifiers in the two translation are different but are not distinct in their significant characters.

Declaration mismatch

Issue

Declaration mismatch occurs when a function or variable declaration does not match other instances of the function or variable.

Risk

When a mismatch occurs between two variable declarations in different compilation units, a typical linker follows an algorithm to pick one declaration for the variable. If you expect a variable declaration that is different from the one chosen by the linker, you can see unexpected results when the variable is used.

A similar issue can occur with mismatch in function declarations.

Fix

The fix depends on the type of declaration mismatch. If both declarations indeed refer to the same object, use the same declaration. If the declarations refer to different objects, change the names of the one of the variables. If you change a variable name, remember to make the change in all places that use the variable.

Sometimes, declaration mismatches can occur because the declarations are affected by previous preprocessing directives. For instance, a declaration occurs in a macro, and the macro is defined on one inclusion path but undefined in another. These declaration mismatches can be tricky to debug. Identify the divergence between the two inclusion paths and fix the conflicting macro definitions.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Inconsistent Declarations in Two Files

file1.c

```
int foo(void) {
    return 1;
}
```

file2.c

```
double foo(void);

int bar(void) {
    return (int)foo();
}
```

In this example, *file1.c* declares `foo()` as returning an integer. In *file2.c*, `foo()` is declared as returning a double. This difference raises a defect on the second instance of `foo` in *file2*.

Correction — Align the Function Return Values

One possible correction is to change the function declarations so that they match. In this example, by changing the declaration of `foo` in *file2.c* to match *file1.c*, the defect is fixed.

file1.c

```
int foo(void) {
    return 1;
}
```

file2.c

```
int foo(void);

int bar(void) {
    return foo();
}
```

Example - Inconsistent Structure Alignment

test1.c	*test2.c*

`#include "square.h"`
`#include "circle.h"`
`struct aCircle circle;`
`struct aSquare square;`
 	
`int main(){`
`  square.side=1;`
`  circle.radius=1;`
`  return 0;`
`}`
	`#include "circle.h"`
	`#include "square.h"`
	`struct aCircle circle;`
	`struct aSquare square;`
	
	`int main(){`
	`  square.side=1;`
	`  circle.radius=1;`
	`  return 0;`
	`}`

circle.h
 	
`#pragma pack(1)`
 	
`extern struct aCircle{`
`  int radius;`
`} circle;`
	square.h
	
	`extern struct aSquare {`
	`  unsigned int side:1;`
	`} square;`

In this example, a declaration mismatch defect is raised on `square` in *square.h* because Polyspace infers that `square` in *square.h* does not have the same alignment as `square` in *test2.c*. This error occurs because the `#pragma pack(1)` statement in *circle.h* declares specific alignment. In *test2.c*, *circle.h* is included before *square.h*. Therefore, the `#pragma pack(1)` statement from *circle.h* is not reset to the default alignment after the `aCircle` structure. Because of this omission, *test2.c* infers that the `aSquare square` structure also has an alignment of `1` byte.

Correction — Close Packing Statements

One possible correction is to reset the structure alignment after the `aCircle` struct declaration. For the GNU or Microsoft Visual compilers, fix the defect by adding a `#pragma pack()` statement at the end of *circle.h*.

test1.c	*test2.c*

`#include "square.h"`
`#include "circle.h"`
`struct aCircle circle;`
`struct aSquare square;`
 	
`int main(){`
`  square.side=1;`
`  circle.radius=1;`
`  return 0;`
`}`
	`#include "circle.h"`
	`#include "square.h"`
	`struct aCircle circle;`
	`struct aSquare square;`
	
	`int main(){`
	`  square.side=1;`
	`  circle.radius=1;`
	`  return 0;`
	`}`

circle.h	*square.h*

`#pragma pack(1)`
 	
`extern struct aCircle{`
`  int radius;`
`} circle;`
 	
`#pragma pack()`
	`extern struct aSquare {`
	`  unsigned int side:1;`
	`} square;`

Other compilers require different `#pragma pack` syntax. For your syntax, see the documentation for your compiler.

Correction — Use the `Ignore pragma pack directives` Option

One possible correction is to add the `Ignore pragma pack directives` option to your Bug Finder analysis. If you want the structure alignment to change for each structure, and you do not want to see this **Declaration mismatch** defect, use this correction.

1. On the Configuration pane, select the **Advanced Settings** pane.
2. In the **Other** box, enter `-ignore-pragma-pack`.
3. Rerun your analysis.

   The **Declaration mismatch** defect is resolved.

Check Information

Decidability: Decidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [insufmem]

Allocating insufficient memory

Description

Rule Definition

Allocating insufficient memory.

Polyspace Implementation

This checker checks for these issues:

- **Wrong allocated object size for cast**.
- **Pointer access out of bounds**.
- **Wrong type used in sizeof**.
- **Possible misuse of sizeof**.

Examples

Wrong allocated object size for cast

Issue

Wrong allocated object size for cast occurs during pointer conversion when the pointer's address is misaligned. If a pointer is converted to a different pointer type, the size of the allocated memory must be a multiple of the size of the destination pointer.

Risk

Dereferencing a misaligned pointer has undefined behavior and can cause your program to crash.

Fix

Suppose you convert a pointer `ptr1` to `ptr2`. If `ptr1` points to a buffer of `N` bytes and `ptr2` is a *`type`* * pointer where `sizeof(`*`type`*`)` is `n` bytes, make sure that `N` is an integer multiple of `n`.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Dynamic Allocation of Pointers

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(13);
    long *dest;

    dest = (long*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to a `long*`. The dynamically allocated memory of `ptr`, 13 bytes, is not a multiple of the size of `dest`, 4 bytes. This misalignment causes the **Wrong allocated object size for cast** defect.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the allocated memory to 12 instead of 13.

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(12);
    long *dest;

    dest = (long*)ptr;
}
```

Example - Static Allocation of Pointers

```
void static_non_align(void){
    char arr[13], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to an `int*` in line 6. `ptr` has a memory size of 13 bytes because the array `arr` has a size of 13 bytes. The size of `dest` is 4 bytes, which is not a multiple of 13. This misalignment causes the **Wrong allocated object size for cast** defect.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the size of the array `arr` to a multiple of 4.

```
void static_non_align(void){
    char arr[12], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr;
}
```

Example - Allocation with a Function

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;
```

```
    dest1 = (int*)my_alloc(13);  //defect
    dest2 = (char*)my_alloc(13); //not a defect
}
```

In this example, the software raises a defect on the conversion of the pointer returned by `my_alloc(13)` to an `int*` in line 11. `my_alloc(13)` returns a pointer with a dynamically allocated size of 13 bytes. The size of `dest1` is 4 bytes, which is not a divisor of 13. This misalignment causes the **Wrong allocated object size for cast** defect. In line 12, the same function call, `my_alloc(13)`, does not call a defect for the conversion to `dest2` because the size of `char*`, 1 byte, a divisor of 13.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the argument for `my_alloc` to a multiple of 4.

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(12);
    dest2 = (char*)my_alloc(13);
}
```

Pointer access out of bounds

Issue

Pointer access out of bounds occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back

using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

Wrong type used in sizeof

Issue

Wrong type used in sizeof occurs when both of the following conditions hold:

- You assign the address of a block of memory to a pointer, or transfer data between two blocks of memory. The assignment or copy uses the `sizeof` operator.

For instance, you initialize a pointer using `malloc(sizeof(`*`type`*`))` or copy data between two addresses using `memcpy(`*`destination_ptr`*`, `*`source_ptr`*`, sizeof(`*`type`*`))`.

- You use an incorrect type as argument of the `sizeof` operator. You use the pointer type instead of the type that the pointer points to.

  For instance, to initialize a *`type`*`*` pointer, you use `malloc(sizeof(`*`type`*`*))` instead of `malloc(sizeof(`*`type`*`))`.

Risk

Irrespective of what *`type`* stands for, the expression `sizeof(`*`type`*`*)` always returns a fixed size. The size returned is the pointer size on your platform in bytes. The appearance of `sizeof(`*`type`*`*)` often indicates an unintended usage. The error can cause allocation of a memory block that is much smaller than what you need and lead to weaknesses such as buffer overflows.

For instance, assume that `structType` is a structure with ten `int` variables. If you initialize a `structType*` pointer using `malloc(sizeof(structType*))` on a 32-bit platform, the pointer is assigned a memory block of four bytes. However, to be allocated completely for one `structType` variable, the `structType*` pointer must point to a memory block of `sizeof(structType) = 10 * sizeof(int)` bytes. The required size is much greater than the actual allocated size of four bytes.

Fix

To initialize a *`type`*`*` pointer, replace `sizeof(`*`type`*`*)` in your pointer initialization expression with `sizeof(`*`type`*`)`.

Example - Allocate a Char Array With `sizeof`

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char*) * 5);
    free(str);

}
```

In this example, memory is allocated for the character pointer `str` using a `malloc` of five char pointers. However, `str` is a pointer to a character, not a pointer to a character pointer. Therefore the `sizeof` argument, `char*`, is incorrect.

Correction — Match Pointer Type to `sizeof` Argument

One possible correction is to match the argument to the pointer type. In this example, `str` is a character pointer, therefore the argument must also be a character.

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char) * 5);
    free(str);

}
```

Possible misuse of sizeof

Issue

Possible misuse of sizeof occurs when Polyspace Bug Finder detects possibly unintended results from the use of `sizeof` operator. For instance:

- You use the `sizeof` operator on an array parameter name, expecting the array size. However, the array parameter name by itself is a pointer. The `sizeof` operator returns the size of that pointer.
- You use the `sizeof` operator on an array element, expecting the array size. However, the operator returns the size of the array element.
- The size argument of certain functions such as `strncmp` or `wcsncpy` is incorrect because you used the `sizeof` operator earlier with possibly incorrect expectations. For instance:
  - In a function call `strncmp(string1, string2, num)`, `num` is obtained from an incorrect use of the `sizeof` operator on a pointer.
  - In a function call `wcsncpy(destination, source, num)`, `num` is the not the number of wide characters but a size in bytes obtained by using the `sizeof` operator. For instance, you use `wcsncpy(destination, source, sizeof(destination) - 1)` instead of `wcsncpy(destination, source, (sizeof(desintation)/sizeof(wchar_t)) - 1)`.

Risk

Incorrect use of the `sizeof` operator can cause the following issues:

- If you expect the `sizeof` operator to return array size and use the return value to constrain a loop, the number of loop runs are smaller than what you expect.
- If you use the return value of `sizeof` operator to allocate a buffer, the buffer size is smaller than what you require. Insufficient buffer can lead to resultant weaknesses such as buffer overflows.
- If you use the return value of `sizeof` operator incorrectly in a function call, the function does not behave as you expect.

Fix

Possible fixes are:

- Do not use the `sizeof` operator on an array parameter name or array element to determine array size.

  The best practice is to pass the array size as a separate function parameter and use that parameter in the function body.
- Use the `sizeof` operator carefully to determine the number argument of functions such as `strncmp` or `wcsncpy`. For instance, for wide string functions such as `wcsncpy`, use the number of wide characters as argument instead of the number of bytes.

Example - `sizeof` Used Incorrectly to Determine Array Size

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < sizeof(a)/sizeof(int); i++)    {
        a[i] = i + 1;
```

```
    }
}
```

In this example, `sizeof(a)` returns the size of the pointer `a` and not the array size.

Correction — Determine Array Size in Another Way

One possible correction is to use another means to determine the array size.

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < MAX_SIZE; i++)    {
        a[i] = i + 1;
    }
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [intoflow]

Overflowing signed integers

Description

Rule Definition

Overflowing signed integers.

Polyspace Implementation

This checker checks for these issues:

- **Integer overflow**.
- **Integer constant overflow**.

Examples

Integer overflow

Issue

Integer overflow occurs when an operation on integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different integer types depends on your processor. See `Target processor type (-target)`.

Risk

Integer overflows on signed integers result in undefined behavior.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

To avoid overflows in general, try one of these techniques:

- Keep integer variable values restricted to within half the range of signed integers.
- In operations that might overflow, check for conditions that can lead to the overflow and implement wrap around or saturation behavior depending on how the result of the operation is used. The result then becomes predictable and can be safely used in subsequent computations.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See "Address Polyspace Results Through Bug Fixes or Justifications".

Example - Addition of Maximum Integer

```
#include <limits.h>

int plusplus(void) {

    int var = INT_MAX;
    var++;
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

Correction — Different Storage Type

One possible correction is to change data types. Store the result of the operation in a larger data type (Note that on a 32-bit machine, `int` and `long` has the same size). In this example, on a 32-bit machine, by returning a `long long` instead of an `int`, the overflow error is fixed.

```
#include <limits.h>

long long plusplus(void) {

    long long lvar = INT_MAX;
    lvar++;
    return lvar;
}
```

Integer constant overflow

Issue

Integer constant overflow occurs when you assign a compile-time constant to a signed integer variable whose data type cannot accommodate the value. An `n`-bit signed integer holds values in the range $[-2^{n-1}, 2^{n-1}-1]$.

For instance, `c` is an 8-bit signed `char` variable that cannot hold the value 255.

```
signed char c = 255;
```

To determine the sizes of fundamental types, Bug Finder uses your specification for `Target processor type (-target)`.

Risk

The default behavior for constant overflows can vary between compilers and platforms. Retaining constant overflows can reduce the portability of your code.

Even if your compilers wraps around overflowing constants with a warning, the wrap-around behavior can be unintended and cause unexpected results.

Fix

Check if the constant value is what you intended. If the value is correct, use a different, possibly wider, data type for the variable.

Example - Overflowing Constant from Macro Expansion

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
    char c1 = MAX_UNSIGNED_CHAR;
    char c2 = MAX_SIGNED_CHAR+1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow. To reproduce these defects, use analysis option `Target processor type (-target)` where `char` is signed by default.

Correction — Use Different Data Type

One possible correction is to use a different data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
    unsigned char c1 = MAX_UNSIGNED_CHAR;
    unsigned char c2 = MAX_SIGNED_CHAR+1;
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [intptrconv]

Converting a pointer to integer or integer to pointer

Description

Rule Definition

Converting a pointer to integer or integer to pointer.

Polyspace Implementation

This checker checks for **Conversion between pointers and integers**.

Examples

Conversion between pointers and integers

Issue

The issue occurs when a conversion is performed between a pointer to object and an integer type.

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Risk

Conversion between integers and pointers can cause errors or undefined behavior.

- If an integer is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.
- If a pointer is cast to an integer, the resulting value can be outside the allowed range for the integer type.

Example - Casts between pointer and integer

```
#include <stdbool.h>

typedef unsigned char      uint8_t;
typedef          char      char_t;
typedef unsigned short     uint16_t;
typedef signed   int       int32_t;

typedef _Bool bool_t;
uint8_t *PORTA = (uint8_t *) 0x0002;            /* Non-compliant */

void foo(void) {

    char_t c = 1;
    char_t *pc = &c;                              /* Compliant */


    uint16_t ui16   = 7U;
    uint16_t *pui16 = &ui16;                      /* Compliant */
    pui16 = (uint16_t *) ui16;                    /* Non-compliant */
```

```
    uint16_t *p;
    int32_t addr = (int32_t) p;                  /* Non-compliant */
    bool_t b = (bool_t) p;                       /* Non-compliant */
    enum etag { A, B } e = ( enum etag ) p;      /* Non-compliant */
}
```

In this example, the rule is violated when:

- The integer `0x0002` is cast to a pointer.

  If the integer defines an absolute address, it is more common to assign the address to a pointer in a header file. To avoid the assignment being flagged, you can then exclude headers files from coding rules checking. For more information, see `Do not generate results for (-do-not-generate-results-for)`.
- The pointer `p` is cast to integer types such as `int32_t`, `bool_t` or `enum etag`.

The rule is not violated when the address `&ui16` is assigned to a pointer.

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [inverrno]

Incorrectly setting and using errno

Description

Rule Definition

Incorrectly setting and using errno.

Polyspace Implementation

This checker checks for these issues:

- **Misuse of errno**.
- **Errno not checked**.
- **Errno not reset**.

Examples

Misuse of errno

Issue

Misuse of errno occurs when you check `errno` for error conditions in situations where checking `errno` does not guarantee the absence of errors. In some cases, checking `errno` can lead to false positives.

For instance, you check `errno` following calls to the functions:

- `fopen`: If you follow the ISO Standard, the function might not set `errno` on errors.
- `atof`: If you follow the ISO Standard, the function does not set `errno`.
- `signal`: The `errno` value indicates an error only if the function returns the `SIG_ERR` error indicator.

Risk

The ISO C Standard does not enforce that these functions set `errno` on errors. Whether the functions set `errno` or not is implementation-dependent.

To detect errors, if you check `errno` alone, the validity of this check also becomes implementation-dependent.

In some cases, the `errno` value indicates an error only if the function returns a specific error indicator. If you check `errno` before checking the function return value, you can see false positives.

Fix

For information on how to detect errors, see the documentation for that specific function.

Typically, the functions return an out-of-band error indicator to indicate errors. For instance:

- `fopen` returns a null pointer if an error occurs.
- `signal` returns the `SIG_ERR` error indicator and sets `errno` to a positive value. Check `errno` only after you have checked the function return value.

Example - Incorrectly Checking for `errno` After `fopen` Call

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define fatal_error() abort()

const char *temp_filename = "/tmp/demo.txt";

FILE *func()
{
    FILE *fileptr;
    errno = 0;
    fileptr = fopen(temp_filename, "w+b");
    if (errno != 0) {
        if (fileptr != NULL) {
            (void)fclose(fileptr);
        }
        /* Handle error */
        fatal_error();
    }
    return fileptr;
}
```

In this example, `errno` is the first variable that is checked after a call to `fopen`. You might expect that `fopen` changes `errno` to a nonzero value if an error occurs. If you run this code with an implementation of `fopen` that does not set `errno` on errors, you might miss an error condition. In this situation, `fopen` can return a null pointer that escapes detection.

Correction — Check Return Value of `fopen` After Call

One possible correction is to only check the return value of `fopen` for a null pointer.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define fatal_error() abort()

const char *temp_filename = "/tmp/demo.txt";

FILE *func()
{
    FILE *fileptr;
    fileptr = fopen(temp_filename, "w+b");
    if (fileptr == NULL) {
        fatal_error();
    }
    return fileptr;
}
```

Errno not checked

Issue

Errno not checked occurs when you call a function that sets `errno` to indicate error conditions, but do not check `errno` after the call. For these functions, checking `errno` is the only reliable way to determine if an error occurred.

Functions that set `errno` on errors include:

- `fgetwc`, `strtol`, and `wcstol`.

  For a comprehensive list of functions, see documentation about errno.
- POSIX `errno`-setting functions such as `encrypt` and `setkey`.

Risk

To see if the function call completed without errors, check `errno` for error values.

The return values of these `errno`-setting functions do not indicate errors. The return value can be one of the following:

- `void`
- Even if an error occurs, the return value can be the same as the value from a successful call. Such return values are called in-band error indicators.

You can determine if an error occurred only by checking `errno`.

For instance, `strtol` converts a string to a long integer and returns the integer. If the result of conversion overflows, the function returns `LONG_MAX` and sets `errno` to `ERANGE`. However, the function can also return `LONG_MAX` from a successful conversion. Only by checking `errno` can you distinguish between an error and a successful conversion.

Fix

Before calling the function, set `errno` to zero.

After the function call, to see if an error occurred, compare `errno` to zero. Alternatively, compare `errno` to known error indicator values. For instance, `strtol` sets `errno` to `ERANGE` to indicate errors.

The error message in the Polyspace result shows the error indicator value that you can compare to.

Example - `errno` Not Checked After Call to `strtol`

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    long val = strtol(str, &endptr, base);
```

```
    printf("Return value of strtol() = %ld\n", val);
}
```

You are using the return value of `strtol` without checking `errno`.

Correction — Check `errno` After Call

Before calling `strtol`, set `errno` to zero . After a call to `strtol`, check the return value for `LONG_MIN` or `LONG_MAX` and `errno` for `ERANGE`.

```
#include<stdlib.h>
#include<stdio.h>
#include<errno.h>
#include<limits.h>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    errno = 0;
    long val = strtol(str, &endptr, base);
    if((val == LONG_MIN || val == LONG_MAX) && errno == ERANGE) {
         printf("strtol error");
         exit(EXIT_FAILURE);
    }
    printf("Return value of strtol() = %ld\n", val);
}
```

Errno not reset

Issue

Errno not reset occurs when you do not reset `errno` before calling a function that sets `errno` to indicate error conditions. However, you check `errno` for those error conditions after the function call.

Risk

The `errno` is not clean and can contain values from a previous call. Checking `errno` for errors can give the false impression that an error occurred.

`errno` is set to zero at program startup but subsequently, `errno` is not reset by a C standard library function. You must explicitly set `errno` to zero when required.

Fix

Before calling a function that sets `errno` to indicate error conditions, reset `errno` to zero explicitly.

Example - `errno` Not Reset Before Call to `strtod`

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>

#define fatal_error() abort()
```

```
double func(const char *s1, const char *s2)
{
    double f1;
    f1 = strtod (s1, NULL);
    if (0 == errno) {
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
                  {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

In this example, `errno` is not reset to 0 before the first call to `strtod`. Checking `errno` for 0 later can lead to a false positive.

Correction — Reset `errno` Before Call

One possible correction is to reset `errno` to 0 before calling `strtod`.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>

#define fatal_error() abort()

double func(const char *s1, const char *s2)
{
    double f1;
    errno = 0;
    f1 = strtod (s1, NULL);
    if (0 == errno) {
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
              {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [invfmtstr]

Using invalid format strings

Description

Rule Definition

Using invalid format strings.

Polyspace Implementation

This checker checks for **Format string specifiers and arguments mismatch**.

Examples

Format string specifiers and arguments mismatch

Issue

Format string specifiers and arguments mismatch occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
  const char *message = "License not available";
  int err_number = ;-4
  printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Printing a Float

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", fst);
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [invptr]

Forming or using out-of-bounds pointers or array subscripts

Description

Rule Definition

Forming or using out-of-bounds pointers or array subscripts.

Polyspace Implementation

This checker checks for these issues:

- **Array access out of bounds**.
- **Pointer access out of bounds**.

Examples

Array access out of bounds

Issue

Array access out of bounds occurs when an array index falls outside the range `[0...array_size-1]` during array access.

Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Array Access Out of Bounds Error

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

Correction — Keep Array Index Within Array Bounds

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

Pointer access out of bounds

Issue

Pointer access out of bounds occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;
```

```
 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [ioileave]

Interleaving stream inputs and outputs without a flush or positioning call

Description

Rule Definition

Interleaving stream inputs and outputs without a flush or positioning call.

Polyspace Implementation

This checker checks for **Alternating input and output from a stream without flush or positioning call**.

Examples

Alternating input and output from a stream without flush or positioning call

Issue

Alternating input and output from a stream without flush or positioning call occurs when:

- You do not perform a flush or function positioning call between an output operation and a following input operation on a file stream in update mode.
- You do not perform a function positioning call between an input operation and a following output operation on a file stream in update mode.

Risk

Alternating input and output operations on a stream without an intervening flush or positioning call is undefined behavior.

Fix

Call `fflush()` or a file positioning function such as `fseek()` or `fsetpos()` between output and input operations on an update stream.

Call a file positioning function between input and output operations on an update stream.

Example - Read After Write Without Intervening Flush

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";

void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;
```

```
    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Read operation after write without
    intervening flush. */
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
          (void)fclose(file);
          /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

In this example, the file `demo.txt` is opened for reading and appending. After the call to `fwrite()`, a call to `fread()` without an intervening flush operation is undefined behavior.

Correction — Call `fflush()` Before the Read Operation

After writing data to the file, before calling `fread()`, perform a flush call.

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";


void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;

    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
```

```
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Buffer flush after write and before read */
    if (fflush(file) != 0)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [liberr]

Failing to detect and handle standard library errors

Description

Rule Definition

Failing to detect and handle standard library errors.

Polyspace Implementation

This checker checks for these issues:

- **Returned value of a sensitive function not checked**.
- **Unprotected dynamic memory allocation**.

Examples

Returned value of a sensitive function not checked

Issue

Returned value of a sensitive function not checked occurs when you call sensitive standard functions, but you:

- Ignore the return value.
- Use an output or a return value without testing the validity of the return value.

For this defect, two type of functions are considered: ***sensitive*** and ***critical sensitive***.

A ***sensitive*** function is a standard function that can encounter:

- Exhausted system resources (for example, when allocating resources)
- Changed privileges or permissions
- Tainted sources when reading, writing, or converting data from external sources
- Unsupported features despite an existing API

A ***critical sensitive*** function is a sensitive function that performs one of these critical or vulnerable tasks:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

Risk

If you do not check the return value of functions that perform sensitive or critical sensitive tasks, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

Fix

Before continuing with the program, test the return value of *critical sensitive* functions.

For *sensitive functions*, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

Example - Sensitive Function Return Ignored

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    pthread_attr_init(&attr);
}
```

This example shows a call to the sensitive function `pthread_attr_init`. The return value of `pthread_attr_init` is ignored, causing a defect.

Correction — Cast Function to `(void)`

One possible correction is to cast the function to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of the sensitive function.

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    (void)pthread_attr_init(&attr);
}
```

Correction — Test Return Value

One possible correction is to test the return value of `pthread_attr_init` to check for errors.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

void initialize() {
    pthread_attr_t attr;
    int result;

    result = pthread_attr_init(&attr);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

Example - Critical Function Return Ignored

```
#include <pthread.h>
extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0));
    pthread_join(thread_id,  &res);
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), Polyspace does not ignore this *Return value of a sensitive function not checked* defect. The other critical function, `pthread_join`, returns value that is ignored implicitly. `pthread_join` uses the return value of `pthread_create`, which was not checked.

Correction — Test the Return Value of Critical Functions

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

Unprotected dynamic memory allocation

Issue

Unprotected dynamic memory allocation occurs when you do not check after dynamic memory allocation whether the memory allocation succeeded.

Risk

When memory is dynamically allocated using `malloc`, `calloc`, or `realloc`, it returns a value `NULL` if the requested memory is not available. If the code following the allocation accesses the memory block without checking for this `NULL` value, this access is not protected from failures.

Fix

Check the return value of `malloc`, `calloc`, or `realloc` for NULL before accessing the allocated memory location.

```
int *ptr = malloc(size * sizeof(int));

if(ptr) /* Check for NULL */
{
   /* Memory access through ptr */
}
```

Example - Unprotected dynamic memory allocation error

```
#include <stdlib.h>

void Assign_Value(void)
{
  int* p = (int*)calloc(5, sizeof(int));

  *p = 2;
  /* Defect: p is not checked for NULL value */

  free(p);
}
```

If the memory allocation fails, the function `calloc` returns `NULL` to `p`. Before accessing the memory through `p`, the code does not check whether `p` is `NULL`

Correction — Check for NULL Value

One possible correction is to check whether `p` has value `NULL` before dereference.

```
#include <stdlib.h>

void Assign_Value(void)
 {
   int* p = (int*)calloc(5, sizeof(int));

   /* Fix: Check if p is NULL */
   if(p!=NULL) *p = 2;

   free(p);
 }
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [libmod]

Modifying the string returned by getenv, localeconv, setlocale, and strerror

Description

Rule Definition

Modifying the string returned by getenv, localeconv, setlocale, and strerror.

Polyspace Implementation

This checker checks for **Modification of internal buffer returned from nonreentrant standard function**.

Examples

Modification of internal buffer returned from nonreentrant standard function

Issue

Modification of internal buffer returned from nonreentrant standard function occurs when the following happens:

- A nonreentrant standard function returns a pointer.
- You attempt to write to the memory location that the pointer points to.

Nonreentrant standard functions that return a non `const`-qualified pointer to an internal buffer include `getenv`, `getlogin`, `crypt`, `setlocale`, `localeconv`, `strerror` and others.

Risk

Modifying the internal buffer that a nonreentrant standard function returns can cause the following issues:

- It is possible that the modification does not succeed or alters other internal data.

  For instance, `getenv` returns a pointer to an environment variable value. If you modify this value, you alter the environment of the process and corrupt other internal data.
- Even if the modification succeeds, it is possible that a subsequent call to the same standard function does not return your modified value.

  For instance, you modify the environment variable value that `getenv` returns. If another process, thread, or signal handler calls `setenv`, the modified value is overwritten. Therefore, a subsequent call to `getenv` does not return your modified value.

Fix

Avoid modifying the internal buffer using the pointer returned from the function.

Example - Modification of `getenv` Return Value

```
#include <stdlib.h>
#include <string.h>
```

```
void printstr(const char*);

void func() {
    char* env = getenv("LANGUAGE");
    if (env != NULL) {
        strncpy(env, "C", 1);
        printstr(env);
    }
}
```

In this example, the first argument of `strncpy` is the return value from a nonreentrant standard function `getenv`. The behavior can be undefined because `strncpy` modifies this argument.

Correction - Copy Return Value of `getenv` and Modify Copy

One possible solution is to copy the return value of `getenv` and pass the copy to the `strncpy` function.

```
#include <stdlib.h>
#include <string.h>
enum {
    SIZE20 = 20
};

void printstr(const char*);

void func() {
    char* env = getenv("LANGUAGE");
    if (env != NULL) {
        char env_cp[SIZE20];
        strncpy(env_cp, env, SIZE20);
        strncpy(env_cp, "C", 1);
        printstr(env_cp);
    }
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [libptr]

Forming invalid pointers by library function

Description

Rule Definition

Forming invalid pointers by library function.

Polyspace Implementation

This checker checks for these issues:

- **Use of path manipulation function without maximum sized buffer checking**.
- **Invalid use of standard library memory routine**.
- **Invalid use of standard library string routine**.
- **Destination buffer overflow in string manipulation**.

Examples

Use of path manipulation function without maximum sized buffer checking

Issue

Use of path manipulation function without maximum-sized buffer checking occurs when the destination argument of a path manipulation function such as `realpath` or `getwd` has a buffer size less than `PATH_MAX` bytes.

Risk

A buffer smaller than `PATH_MAX` bytes can overflow but you cannot test the function return value to determine if an overflow occurred. If an overflow occurs, following the function call, the content of the buffer is undefined.

For instance, `char *getwd(char *buf)` copies an absolute path name of the current folder to its argument. If the length of the absolute path name is greater than `PATH_MAX` bytes, `getwd` returns `NULL` and the content of `*buf` is undefined. You can test the return value of `getwd` for `NULL` to see if the function call succeeded.

However, if the allowed buffer for `buf` is less than `PATH_MAX` bytes, a failure can occur for a smaller absolute path name. In this case, `getwd` does not return `NULL` even though a failure occurred. Therefore, the allowed buffer for `buf` must be `PATH_MAX` bytes long.

Fix

Possible fixes are:

- Use a buffer size of `PATH_MAX` bytes. If you obtain the buffer from an unknown source, before using the buffer as argument of `getwd` or `realpath` function, make sure that the size is less than `PATH_MAX` bytes.

- Use a path manipulation function that allows you to specify a buffer size.

  For instance, if you are using `getwd` to get the absolute path name of the current folder, use `char *getcwd(char *buf, size_t size);` instead. The additional argument `size` allows you to specify a size greater than or equal to `PATH_MAX`.
- Allow the function to allocate additional memory dynamically, if possible.

  For instance, `char *realpath(const char *path, char *resolved_path);` dynamically allocates memory if `resolved_path` is `NULL`. However, you have to deallocate this memory later using the `free` function.

Example - Possible Buffer Overflow in Use of `getwd` Function

```
#include <unistd.h>
#include <linux/limits.h>
#include <stdio.h>

void func(void) {
    char buf[PATH_MAX];
    if (getwd(buf+1)!= NULL)         {
        printf("cwd is %s\n", buf);
    }
}
```

In this example, although the array `buf` has `PATH_MAX` bytes, the argument of `getwd` is `buf + 1`, whose allowed buffer is less than `PATH_MAX` bytes.

Correction — Use Array of Size `PATH_MAX` Bytes

One possible correction is to use an array argument with size equal to `PATH_MAX` bytes.

```
#include <unistd.h>
#include <linux/limits.h>
#include <stdio.h>

void func(void) {
    char buf[PATH_MAX];
    if (getwd(buf)!= NULL)         {
        printf("cwd is %s\n", buf);
    }
}
```

Invalid use of standard library memory routine

Issue

Invalid use of standard library memory routine occurs when a memory library function is called with invalid arguments. For instance, the `memcpy` function copies to an array that cannot accommodate the number of bytes copied.

Risk

Use of a memory library function with invalid arguments can result in issues such as buffer overflow.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do

not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Invalid Use of Standard Library Memory Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string `str2` is 5, but six characters of string `str1` are copied into `str2` using the `memcpy` function.

Correction — Call Function with Valid Arguments

One possible correction is to adjust the size of `str2` so that it accommodates the characters copied with the `memcpy` function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  return str2;
 }
```

Invalid use of standard library string routine

Issue

Invalid use of standard library string routine occurs when a string library function is called with invalid arguments.

Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Invalid Use of Standard Library String Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_String(void)
{
 char *res;
 char gbuffer[5],text[20]="ABCDEFGHIJKL";

 res=strcpy(gbuffer,text);
 /* Error: Size of text is less than gbuffer */

 return(res);
}
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
#include <stdio.h>

char* Copy_String(void)
{
 char *res;
 /*Fix: gbuffer has equal or larger size than text */
 char gbuffer[20],text[20]="ABCDEFGHIJKL";

 res=strcpy(gbuffer,text);

 return(res);
}
```

Destination buffer overflow in string manipulation

Issue

Destination buffer overflow in string manipulation occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

Example - Buffer Overflow in `sprintf` Use

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [libuse]

Using an object overwritten by getenv, localeconv, setlocale, and strerror

Description

Rule Definition

Using an object overwritten by getenv, localeconv, setlocale, and strerror.

Polyspace Implementation

This checker checks for **Misuse of return value from nonreentrant standard function**.

Examples

Misuse of return value from nonreentrant standard function

Issue

Misuse of return value from nonreentrant standard function occurs when these events happen in this sequence:

1. You point to the buffer returned from a nonreentrant standard function such as `getenv` or `setlocale`.

   ```
   user = getenv("USER");
   ```

2. You call that nonreentrant standard function again.

   ```
   user2 = getenv("USER2");
   ```

3. You use or dereference the pointer from the first step expecting the buffer to remain unmodified since that step. In the meantime, the call in the second step has modified the buffer.

   For instance:

   ```
   var=*user;
   ```

In some cases, the defect might appear even if you do not call the `getenv` function a second time but simply return the pointer. For instance:

```
char* func() {
     user=getenv("USER");
     .
     .
     return user;
}
```

For information on which functions are covered by this defect, see documentation on nonreentrant standard functions.

Risk

The C Standard allows nonreentrant functions such as `getenv` to return a pointer to a *static* buffer. Because the buffer is static, a second call to `getenv` modifies the buffer. If you continue to use the

pointer returned from the first call past the second call, you can see unexpected results. The buffer that it points to no longer has values from the first call.

The defect appears even if you do not call `getenv` a second time but simply return the pointer. The reason is that someone calling your function might use the returned pointer *after* a second call to `getenv`. By returning the pointer from your call to `getenv`, you make your function unsafe to use.

The same rationale is true for other nonreentrant functions covered by this defect.

Fix

After the first call to `getenv`, make a copy of the buffer that the returned pointer points to. After the second call to `getenv`, use this copy. Even if the second call modifies the buffer, your copy is untouched.

Example - Return from getenv Used After Second Call to getenv

```
#include <stdlib.h>
#include <string.h>

int func()
{
    int result = 0;

    char *home = getenv("HOME");   /* First call */
    if (home != NULL) {
        char *user = NULL;
        char *user_name_from_home = strrchr(home, '/');

        if (user_name_from_home != NULL) {
            user = getenv("USER");   /* Second call */
            if ((user != NULL) &&
                (strcmp(user, user_name_from_home) == 0))
            {
                result = 1;
            }
        }
    }
    return result;
}
```

In this example, the pointer `user_name_from_home` is derived from the pointer `home`. `home` points to the buffer returned from the first call to `getenv`. Therefore, `user_name_from_home` points to a location in the same buffer.

After the second call to `getenv`, the buffer is modified. If you continue to use `user_name_from_home`, you can get unexpected results.

Correction — Make Copy of Buffer Before Second Call

If you want to access the buffer from the first call to `getenv` past the second call, make a copy of the buffer after the first call. One possible correction is to use the `strdup` function to make the copy.

```
#include <stdlib.h>
#include <string.h>

int func()
{
```

```
    int result = 0;

    char *home = getenv("HOME");
    if (home != NULL) {
        char *user = NULL;
        char *user_name_from_home = strrchr(home, '/');
        if (user_name_from_home != NULL) {
            /* Make copy before second call */
            char *saved_user_name_from_home = strdup(user_name_from_home);
            if (saved_user_name_from_home != NULL) {
                user = getenv("USER");
                if ((user != NULL) &&
                    (strcmp(user, saved_user_name_from_home) == 0))
                {
                    result = 1;
                }
                free(saved_user_name_from_home);
            }
        }
    }
    return result;
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [nonnullcs]

Passing a non-null-terminated character sequence to a library function

Description

Rule Definition

Passing a non-null-terminated character sequence to a library function.

Polyspace Implementation

This checker checks for **Invalid use of standard library string routine**.

Examples

Invalid use of standard library string routine

Issue

Invalid use of standard library string routine occurs when a string library function is called with invalid arguments.

Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Invalid Use of Standard Library String Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_String(void)
{
 char *res;
 char gbuffer[5],text[20]="ABCDEFGHIJKL";
```

```
  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [nullref]

Dereferencing an out-of-domain pointer

Description

Rule Definition

Dereferencing an out-of-domain pointer.

Polyspace Implementation

This checker checks for these issues:

- **Unsafe pointer arithmetic**.
- **Invalid use of standard library memory routine**.
- **Null pointer**.
- **Arithmetic operation with NULL pointer**.
- **Invalid use of standard library string routine**.
- **Use of tainted pointer**.

Examples

Unsafe pointer arithmetic

Issue

The issue occurs when a pointer resulting from arithmetic on a pointer operand does not address an element of the same array as that pointer operand.

Polyspace flags this rule during the analysis as:

- Bug Finder — `Array access out-of-bounds` and `Pointer access out-of-bounds`
- Code Prover — and

Bug Finder and Code Prover check this rule differently and can show different results for this rule. In Code Prover, you can also see a difference in results based on your choice for the option `Verification level (-to)`. See “Check for Coding Standard Violations”.

Risk

Using an invalid array subscript can lead to erroneous behavior of the program. Run-time derived array subscripts are especially troublesome because they cannot be easily checked by manual review or static analysis.

The C Standard defines the creation of a pointer to one beyond the end of the array. The rule permits the C Standard. Dereferencing a pointer to one beyond the end of an array causes undefined behavior and is noncompliant.

Invalid use of standard library memory routine

Issue

Invalid use of standard library memory routine occurs when a memory library function is called with invalid arguments. For instance, the `memcpy` function copies to an array that cannot accommodate the number of bytes copied.

Risk

Use of a memory library function with invalid arguments can result in issues such as buffer overflow.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Invalid Use of Standard Library Memory Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string `str2` is 5, but six characters of string `str1` are copied into `str2` using the `memcpy` function.

Correction — Call Function with Valid Arguments

One possible correction is to adjust the size of `str2` so that it accommodates the characters copied with the `memcpy` function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];

  printf("Enter string:\n");
```

```
  scanf("%s",str1);

  memcpy(str2,str1,6);
  return str2;
 }
```

Null pointer

Issue

Null pointer occurs when you use a pointer with a value of `NULL` as if it points to a valid memory location.

Risk

Dereferencing a null pointer is undefined behavior. In most implementations, the dereference can cause your program to crash.

Fix

Check a pointer for `NULL` before dereference.

If the issue occurs despite an earlier check for `NULL`, look for intermediate events between the check and the subsequent dereference. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

Example - Null pointer error

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 int* p=NULL;

 *p=arr[0];
 /* Defect: Null pointer dereference */

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

The pointer `p` is initialized with value of `NULL`. However, when the value `arr[0]` is written to `*p`, `p` is assumed to point to a valid memory location.

Correction — Assign Address to Null Pointer Before Dereference

One possible correction is to initialize `p` with a valid memory address before dereference.

```
#include <stdlib.h>
```

```
int FindMax(int *arr, int Size)
{
 /* Fix: Assign address to null pointer */
 int* p=&arr[0];

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

Arithmetic operation with NULL pointer

Issue

Arithmetic operation with NULL pointer occurs when an arithmetic operation involves a pointer whose value is `NULL`.

Risk

Performing pointer arithmetic on a null pointer and dereferencing the resulting pointer is undefined behavior. In most implementations, the dereference can cause your program to crash.

Fix

Check a pointer for `NULL` before arithmetic operations on the pointer.

If the issue occurs despite an earlier check for `NULL`, look for intermediate events between the check and the subsequent dereference. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

Example - Arithmetic Operation with NULL Pointer Error

```
#include<stdlib.h>

int Check_Next_Value(int *loc, int val)
 {
  int *ptr = loc, found = 0;

  if (ptr==NULL)
   {
      ptr++;
      /* Defect: NULL pointer shifted */

      if (*ptr==val) found=1;
   }

  return(found);
 }
```

When `ptr` is a `NULL` pointer, the code enters the `if` statement body. Therefore, a `NULL` pointer is shifted in the statement `ptr++`.

Correction — Avoid NULL Pointer Arithmetic

One possible correction is to perform the arithmetic operation when `ptr` is not `NULL`.

```
#include<stdlib.h>

int Check_Next_Value(int *loc, int val)
 {
  int *ptr = loc, found = 0;

  /* Fix: Perform operation when ptr is not NULL */
  if (ptr!=NULL)
   {
      ptr++;

      if (*ptr==val) found=1;
   }

  return(found);
 }
```

Invalid use of standard library string routine

Issue

Invalid use of standard library string routine occurs when a string library function is called with invalid arguments.

Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Invalid Use of Standard Library String Routine Error

```
 #include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";
```

```
  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

Use of tainted pointer

Issue

Use of tainted pointer defect is raised when:

- Tainted NULL pointer — the pointer is not validated against NULL.
- Tainted size pointer — the size of the memory zone that a pointer points to is not validated.

Note On a single pointer, your code can have instances of **Use of tainted pointer**, **Pointer dereference with tainted offset**, and **Tainted NULL or non-null-terminated string**. Bug Finder raises only the first tainted pointer defect that it finds.

Risk

An attacker can give your program a pointer that points to unexpected memory locations. If the pointer is dereferenced to write, the attacker can:

- Modify the state variables of a critical program.
- Cause your program to crash.
- Execute unwanted code.

If the pointer is dereferenced to read, the attacker can:

- Read sensitive data.

- Cause your program to crash.
- Modify a program variable to an unexpected value.

Fix

Avoid use of pointers from external sources.

Alternatively, if you trust the external source, sanitize the pointer before dereference. In a separate sanitization function:

- Check that the pointer is not NULL.
- Check the size of the memory location (if possible). This second check validates whether the size of the data the pointer points to matches the size your program expects.

The defect still appears in the body of the sanitization function. However, if you use a sanitization function, instead of several occurrences, the defect appears only once. You can justify the defect and hide it in later reviews by using code annotations. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Function That Dereferences an External Pointer

```
void taintedptr(int* p, int i) {
    *p = i;
}
```

In this example, the pointer `*p` is passed as an argument, and the value is changed. The pointer can be null or point to unknown memory, which can be vulnerable.

Correction — Avoid Use of External Pointers

One possible correction is to avoid pointers from external sources.

```
int *taintedptr(int i) {
    /* Use heap memory allocated in the application */
    int *p = (int *)malloc(sizeof (int));
    if (p != NULL) { /* Check for success */
        *p = i;
    }
return p;
}
```

Correction — Check Pointer

Another possible correction is to sanitize the pointer before using it. This example uses a second function to check if the pointer is null and can be dereferenced.

```
#include <stdlib.h>

int* sanitize_ptr(int* p) {
    int* res = NULL;
    if (p && *p) { /* Tainted pointer detected here, used as "firewall" */
        /* Pointer is not null and dereference ok */
        res = p;
    }
    return res;
}
void taintedptr(int* p, int i) {
```

```
    p = sanitize_ptr(p);
    if (p) {
        *p = i;
    }
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [padcomp]

Comparison of padding data

Description

Rule Definition

Comparison of padding data.

Polyspace Implementation

This checker checks for **Memory comparison of padding data**.

Examples

Memory comparison of padding data

Issue

Memory comparison of padding data occurs when you use the `memcmp` function to compare two structures as a whole. In the process, you compare meaningless data stored in the structure padding.

For instance:

```
struct structType {
    char member1;
    int member2;
    .
    .
};

structType var1;
structType var2;
.
.
if(memcmp(&var1,&var2,sizeof(var1)))
{...}
```

Risk

If members of a structure have different data types, your compiler introduces additional padding for data alignment in memory. For an example of padding, see `Higher Estimate of Local Variable Size`.

The content of these extra padding bytes is meaningless. The C Standard allows the content of these bytes to be indeterminate, giving different compilers latitude to implement their own padding. If you perform a byte-by-byte comparison of structures with `memcmp`, you compare even the meaningless data stored in the padding. You might reach the false conclusion that two data structures are not equal, even if their corresponding members have the same value.

Fix

Instead of comparing two structures in one attempt, compare the structures member by member.

For efficient code, write a function that does the comparison member by member. Use this function for comparing two structures.

You can use `memcmp` for byte-by-byte comparison of structures only if you know that the structures do not contain padding. Typically, to prevent padding, you use specific attributes or pragmas such as `#pragma pack`. However, these attributes or pragmas are not supported by all compilers and make your code implementation-dependent. If your structures contain bit-fields, using these attributes or pragmas cannot prevent padding.

Example - Structures Compared with `memcmp`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{

    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    if (0 == memcmp(left, right, sizeof(S_Padding)))
    {
        return 1;
    }
    else
        return 0;
}
```

In this example, `memcmp` compares byte-by-byte the two structures that `left` and `right` point to. Even if the values stored in the structure members are the same, the comparison can show an inequality if the meaningless values in the padding bytes are not the same.

Correction — Compare Structures Member by Member

One possible correction is to compare individual structure members.

Note You can compare entire arrays by using `memcmp`. All members of an array have the same data type. Padding bytes are not required to store arrays.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{
    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    return ((left->c == right->c) &&
            (left->i == right->i) &&
            (left->bf1 == right->bf1) &&
            (left->bf2 == right->bf2) &&
            (memcmp(left->buffer, right->buffer, 20) == 0));
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [ptrcomp]

Accessing an object through a pointer to an incompatible type

Description

Rule Definition

Accessing an object through a pointer to an incompatible type.

Polyspace Implementation

This checker checks for **Conversion between pointers to different objects**.

Examples

Conversion between pointers to different objects

Issue

The issue occurs when a cast is performed between a pointer to object type and a pointer to a different object type.

Risk

If a pointer to an object is cast into a pointer to a different object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.

Even if the conversion produces a pointer that is correctly aligned, the behavior can be undefined if the pointer is used to access an object.

Exception: You can convert a pointer to object type into a pointer to one of the following types:

- `char`
- `signed char`
- `unsigned char`

Example - Noncompliant: Cast to Pointer Pointing to Object of Wider Type

```
signed   char *p1;
unsigned int *p2;

void foo(void){
  p2 = ( unsigned int * ) p1;     /* Non-compliant */
}
```

In this example, `p1` can point to a `signed char` object. However, `p1` is cast to a pointer that points to an object of wider type, `unsigned int`.

Example - Noncompliant: Cast to Pointer Pointing to Object of Narrower Type

```
extern unsigned int read_value ( void );
extern void display ( unsigned int n );
```

```
void foo ( void ){
  unsigned int u = read_value ( );
  unsigned short *hi_p = ( unsigned short * ) &u;    /* Non-compliant  */
  *hi_p = 0;
  display ( u );
}
```

In this example, `u` is an `unsigned int` variable. `&u` is cast to a pointer that points to an object of narrower type, `unsigned short`.

On a big-endian machine, the statement `*hi_p = 0` attempts to clear the high bits of the memory location that `&u` points to. But, from the result of `display(u)`, you might find that the high bits have not been cleared.

Example - Compliant: Cast Adding a Type Qualifier

```
const short *p;
const volatile short *q;
void foo (void){
  q = ( const volatile short * ) p;  /* Compliant */
}
```

In this example, both `p` and `q` can point to `short` objects. The cast between them adds a `volatile` qualifier only and is therefore compliant.

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [ptrobj]

Subtracting or comparing two pointers that do not refer to the same array

Description

Rule Definition

Subtracting or comparing two pointers that do not refer to the same array.

Polyspace Implementation

This checker checks for **Subtraction or comparison between pointers to different arrays**.

Examples

Subtraction or comparison between pointers to different arrays

Issue

Subtraction or comparison between pointers to different arrays occurs when you subtract or compare pointers that are null or that point to elements in different arrays. The relational operators for the comparison are >, <, >=, and <=.

Risk

When you subtract two pointers to elements in the same array, the result is the difference between the subscripts of the two array elements. Similarly, when you compare two pointers to array elements, the result is the positions of the pointers relative to each other. If the pointers are null or point to different arrays, a subtraction or comparison operation is undefined. If you use the subtraction result as a buffer index, it can cause a buffer overflow.

Fix

Before you subtract or use relational operators to compare pointers to array elements, check that they are non-null and that they point to the same array.

Example - Subtraction Between Pointers to Elements in Different Arrays

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE20 20

size_t func(void)
{
    int nums[SIZE20];
    int end;
    int *next_num_ptr = nums;
    size_t free_elements;
    /* Increment next_num_ptr as array fills */

    /* Subtraction operation is undefined unless array nums
    is adjacent to variable end in memory. */
```

```
    free_elements = &end - next_num_ptr;
    return free_elements;
}
```

In this example, the array `nums` is incrementally filled. Pointer subtraction is then used to determine how many free elements remain. Unless `end` points to a memory location one past the last element of `nums`, the subtraction operation is undefined.

Correction — Subtract Pointers to the Same Array

Subtract the pointer to the last element that was filled from the pointer to the last element in the array.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE20 20

size_t func(void)
{
    int nums[SIZE20];
    int *next_num_ptr = nums;
    size_t free_elements;
    /* Increment next_num_ptr as array fills */

    /* Subtraction operation involves pointers to the same array. */
    free_elements = &(nums[SIZE20 - 1]) - next_num_ptr;

    return free_elements + 1;
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [resident]

Using identifiers that are reserved for the implementation

Description

Rule Definition

Using identifiers that are reserved for the implementation.

Polyspace Implementation

This checker checks for **Declaration of reserved identifiers or macro names**.

Examples

Declaration of reserved identifiers or macro names

Issue

The issue occurs when a reserved identifier or macro name is declared.

If you define a macro name that corresponds to a standard library macro, object, or function, rule 21.1 is violated.

The rule considers tentative definitions as definitions.

Risk

The Standard allows implementations to treat reserved identifiers specially. If you reuse reserved identifiers, you can cause undefined behavior.

Check Information

Decidability: Decidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [restrict]

Passing pointers into the same object as arguments to different restrict-qualified parameters

Description

Rule Definition

Passing pointers into the same object as arguments to different restrict-qualified parameters.

Polyspace Implementation

This checker checks for **Copy of overlapping memory**.

Examples

Copy of overlapping memory

Issue

Copy of overlapping memory occurs when there is a memory overlap between the source and destination argument of a copy function such as `memcpy` or `strcpy`. For instance, the source and destination arguments of `strcpy` are pointers to different elements in the same string.

Risk

If there is memory overlap between the source and destination arguments of copy functions, according to C standards, the behavior is undefined.

Fix

Determine if the memory overlap is what you want. If so, find an alternative function. For instance:

- If you are using `memcpy` to copy values from one memory location to another, use `memmove` instead of `memcpy`.
- If you are using `strcpy` to copy one string to another, use `memmove` instead of `strcpy`, as follows:

  ```
  s = strlen(source);
  memmove(destination, source, s + 1);
  ```

  `strlen` determines the string length without the null terminator. Therefore, you must move `s+1` bytes instead of `s` bytes.

Example - Overlapping Copy

```
#include <string.h>

char str[] = {"ABCDEFGH"};

void my_copy() {
    strcpy(&str[0],(const char*)&str[2]);
}
```

In this example, because the source and destination argument are pointers to the same string `str`, there is memory overlap between their allowed buffers.

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [sigcall]

Calling signal from interruptible signal handlers

Description

Rule Definition

Calling signal from interruptible signal handlers.

Polyspace Implementation

This checker checks for **Signal call from within signal handler**.

Examples

Signal call from within signal handler

Issue

Signal call from within signal handler occurs when you call `signal()` from a nonpersistent signal handler on a Windows platform.

Risk

A nonpersistent signal handler is reset after catching a signal. The handler does not catch subsequent signals unless the handler is reestablished by calling `signal()`. A nonpersistent signal handler on a Windows platform is reset to `SIG_DFL`. If another signal interrupts the execution of the handler, that signal can cause a race condition between `SIG_DFL` and the existing signal handler. A call to `signal()` can also result in an infinite loop inside the handler.

Fix

Do not call `signal()` from a signal handler on Windows platforms.

Example - `signal()` Called from Signal Handler

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;

void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;

    /* Call signal() to reestablish sig_handler
    upon receiving SIG_ERR. */

    if (signal(s0, sig_handler) == SIG_ERR)
```

```
    {
        /* Handle error */
    }
}

void func(void)
{
        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */

        }
  /* more code */
}
```

In this example, the definition of `sig_handler()` includes a call to `signal()` when the handler catches SIG_ERR. On Windows platforms, signal handlers are nonpersistent. This code can result in a race condition.

Correction — Do Not Call `signal()` from Signal Handler

If your code requires the use of a persistent signal handler on a Windows platform, use a persistent signal handler after performing a thorough risk analysis.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;


void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;
    /* No call to signal() */
}

int main(void)
{

        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */

        }
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [signconv]

Conversion of signed characters to wider integer types before a check for EOF

Description

Rule Definition

Conversion of signed characters to wider integer types before a check for EOF.

Polyspace Implementation

This checker checks for **Misuse of sign-extended character value**.

Examples

Misuse of sign-extended character value

Issue

Misuse of sign-extended character value occurs when you convert a signed or plain `char` data type to a wider integer data type with sign extension. You then use the resulting sign-extended value as array index, for comparison with EOF or as argument to a character-handling function.

Risk

Comparison with EOF: Suppose, your compiler implements the plain `char` type as signed. In this implementation, the character with the decimal form of 255 (–1 in two's complement form) is stored as a signed value. When you convert a `char` variable to the wider data type `int` for instance, the sign bit is preserved (sign extension). This sign extension results in the character with the decimal form 255 being converted to the integer –1, which cannot be distinguished from EOF.

Use as array index: By similar reasoning, you cannot use sign-extended plain `char` variables as array index. If the sign bit is preserved, the conversion from `char` to `int` can result in negative integers. You must use positive integer values for array index.

Argument to character-handling function: By similar reasoning, you cannot use sign-extended plain `char` variables as arguments to character-handling functions declared in `ctype.h`, for instance, `isalpha()` or `isdigit()`. According to the C11 standard (Section 7.4), if you supply an integer argument that cannot be represented as `unsigned char` or EOF, the resulting behavior is undefined.

Fix

Before conversion to a wider integer data type, cast the signed or plain `char` value explicitly to `unsigned char`.

Example - Sign-Extended Character Value Compared with EOF

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];
```

```
static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = *buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

In this example, the function `parser` can traverse a string input `buf`. If a character in the string has the decimal form 255, when converted to the `int` variable `c`, its value becomes –1, which is indistinguishable from `EOF`. The later comparison with `EOF` can lead to a false positive.

Correction — Cast to `unsigned char` Before Conversion

One possible correction is to cast the plain `char` value to `unsigned char` before conversion to the wider `int` type.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];

static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = (unsigned char)*buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [sizeofptr]

Taking the size of a pointer to determine the size of the pointed-to type

Description

Rule Definition

Taking the size of a pointer to determine the size of the pointed-to type.

Polyspace Implementation

This checker checks for **Possible misuse of sizeof**.

Examples

Possible misuse of sizeof

Issue

Possible misuse of sizeof occurs when Polyspace Bug Finder detects possibly unintended results from the use of `sizeof` operator. For instance:

- You use the `sizeof` operator on an array parameter name, expecting the array size. However, the array parameter name by itself is a pointer. The `sizeof` operator returns the size of that pointer.
- You use the `sizeof` operator on an array element, expecting the array size. However, the operator returns the size of the array element.
- The size argument of certain functions such as `strncmp` or `wcsncpy` is incorrect because you used the `sizeof` operator earlier with possibly incorrect expectations. For instance:
  - In a function call `strncmp(string1, string2, num)`, `num` is obtained from an incorrect use of the `sizeof` operator on a pointer.
  - In a function call `wcsncpy(destination, source, num)`, `num` is the not the number of wide characters but a size in bytes obtained by using the `sizeof` operator. For instance, you use `wcsncpy(destination, source, sizeof(destination) - 1)` instead of `wcsncpy(destination, source, (sizeof(desintation)/sizeof(wchar_t)) - 1)`.

Risk

Incorrect use of the `sizeof` operator can cause the following issues:

- If you expect the `sizeof` operator to return array size and use the return value to constrain a loop, the number of loop runs are smaller than what you expect.
- If you use the return value of `sizeof` operator to allocate a buffer, the buffer size is smaller than what you require. Insufficient buffer can lead to resultant weaknesses such as buffer overflows.
- If you use the return value of `sizeof` operator incorrectly in a function call, the function does not behave as you expect.

Fix

Possible fixes are:

- Do not use the `sizeof` operator on an array parameter name or array element to determine array size.

  The best practice is to pass the array size as a separate function parameter and use that parameter in the function body.
- Use the `sizeof` operator carefully to determine the number argument of functions such as `strncmp` or `wcsncpy`. For instance, for wide string functions such as `wcsncpy`, use the number of wide characters as argument instead of the number of bytes.

Example - `sizeof` Used Incorrectly to Determine Array Size

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < sizeof(a)/sizeof(int); i++)    {
        a[i] = i + 1;
    }
}
```

In this example, `sizeof(a)` returns the size of the pointer `a` and not the array size.

Correction — Determine Array Size in Another Way

One possible correction is to use another means to determine the array size.

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < MAX_SIZE; i++)    {
        a[i] = i + 1;
    }
}
```

Check Information

Decidability: Decidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [strmod]

Modifying string literals

Description

Rule Definition

Modifying string literals.

Polyspace Implementation

This checker checks for **Writing to const qualified object**.

Examples

Writing to const qualified object

Issue

Writing to `const` qualified object occurs when you do one of the following:

- Use a `const`-qualified object as the destination of an assignment.
- Pass a `const`-qualified object to a function that modifies the argument.

For instance, the defect can occur in the following situations:

- You pass a `const`-qualified object as first argument of one of the following functions:
  - `mkstemp`
  - `mkostemp`
  - `mkostemps`
  - `mkdtemp`
- You pass a `const`-qualified object as the destination argument of one of the following functions:
  - `strcpy`
  - `strncpy`
  - `strcat`
  - `memset`
- You perform a write operation on a `const`-qualified object.

Risk

The risk depends upon the modifications made to the `const`-qualified object.

Situation	Risk
Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on.	These functions replace the last six characters of their first argument with a string. Therefore, they expect a modifiable `char` array as their first argument.
Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on.	These functions modify their destination argument. Therefore, they expect a modifiable `char` array as their destination argument.
Writing to the object	The `const` qualifier implies an agreement that the value of the object will not be modified. By writing to a `const`-qualified object, you break the agreement. The result of the operation is undefined.

Fix

The fix depends on the modification made to the `const`-qualified object.

Situation	Fix
Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on.	Pass a non-`const` object as first argument of the function.
Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on.	Pass a non-`const` object as destination argument of the function.
Writing to the object	Perform the write operation on a non-`const` object.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Writing to `const`-Qualified Object

```
#include <string.h>

const char* buffer = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

In this example, because `buffer` is `const`-qualified, `strchr(buffer,'X')` returns a `const`-qualified `char*` pointer. When this `char*` pointer is used as the destination argument of `strcpy`, a **Writing to const qualified object** error appears.

Correction — Copy `const`-Qualified Object to Non-`const` Object

One possible correction is to assign the constant string to a non-`const` object and use the non-`const` object as destination argument of `strchr`.

```
#include <string.h>
```

```
char buffer[] = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [swtchdflt]

Use of an implied default in a switch statement

Description

Rule Definition

Use of an implied default in a switch statement.

Polyspace Implementation

This checker checks for **Missing case for switch condition**.

Examples

Missing case for switch condition

Issue

Missing case for switch condition occurs when the `switch` variable can take values that are not covered by a `case` statement.

Note Bug Finder only raises a defect if the switch variable is not full range.

Risk

If the `switch` variable takes a value that is not covered by a `case` statement, your program can have unintended behavior.

A switch-statement that makes a security decision is particularly vulnerable when all possible values are not explicitly handled. An attacker can use this situation to deviate the normal execution flow.

Fix

It is good practice to use a `default` statement as a catch-all for values that are not covered by a `case` statement. Even if the `switch` variable takes an unintended value, the resulting behavior can be anticipated.

Example - Missing Default Condition

```
#include <stdio.h>
#include <string.h>

typedef enum E
{
    ADMIN=1,
    GUEST,
    UNKNOWN = 0
} LOGIN;

static LOGIN system_access(const char *username) {
```

```
  LOGIN user = UNKNOWN;

  if ( strcmp(username, "root") == 0 )
    user = ADMIN;

  if ( strcmp(username, "friend") == 0 )
    user = GUEST;

  return user;
}

int identify_bad_user(const char * username)
{
    int r=0;

    switch( system_access(username) )
    {
    case ADMIN:
        r = 1;
        break;
    case GUEST:
        r = 2;
    }

    printf("Welcome!\n");
    return r;
}
```

In this example, the `enum` parameter `User` can take a value `UNKNOWN` that is not covered by a `case` statement.

Correction — Add a Default Condition

One possible correction is to add a default condition for possible values that are not covered by a `case` statement.

```
#include <stdio.h>
#include <string.h>

typedef enum E
{
    ADMIN=1,
    GUEST,
    UNKNOWN = 0
} LOGIN;

static LOGIN system_access(const char *username) {
  LOGIN user = UNKNOWN;

  if ( strcmp(username, "root") == 0 )
    user = ADMIN;

  if ( strcmp(username, "friend") == 0 )
    user = GUEST;

  return user;
}
```

```
int identify_bad_user(const char * username)
{
    int r=0;

    switch( system_access(username) )
    {
    case ADMIN:
        r = 1;
        break;
    case GUEST:
        r = 2;
    break;
    default:
        printf("Invalid login credentials!\n");
    }

    printf("Welcome!\n");
    return r;
}
```

Check Information

Decidability: Decidable

See Also

Check ISO/IEC TS 17961 (-iso-17961)

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [syscall]

Calling system

Description

Rule Definition

Calling system.

Polyspace Implementation

This checker checks for **Unsafe call to a system function**.

Examples

Unsafe call to a system function

Issue

Unsafe call to a system function occurs when you use a function that invokes an implementation-defined command processor. These functions include:

- The C standard `system()` function.
- The POSIX `popen()` function.
- The Windows `_popen()` and `_wpopen()` functions.

Risk

If the argument of a function that invokes a command processor is not sanitized, it can cause exploitable vulnerabilities. An attacker can execute arbitrary commands or read and modify data anywhere on the system.

Fix

Do not use a `system`-family function to invoke a command processor. Instead, use safer functions such as POSIX `execve()` and WinAPI `CreateProcess()`.

Example - `system()` Called

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};

void func(char *arg)
{
    char buf[SIZE512];
    int retval=sprintf(buf, "/usr/bin/any_cmd %s", arg);
```

```
    if (retval<=0 || retval>SIZE512){
        /* Handle error */
        abort();
    }
    /* Use of system() to pass any_cmd with
    unsanitized argument to command processor */

    if (system(buf) == -1) {
    /* Handle error */
  }
}
```

In this example, `system()` passes its argument to the host environment for the command processor to execute. This code is vulnerable to an attack by command-injection.

Correction — Sanitize Argument and Use `execve()`

In the following code, the argument of `any_cmd` is sanitized, and then passed to `execve()` for execution. `exec`-family functions are not vulnerable to command-injection attacks.

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};


void func(char *arg)
{
  char *const args[SIZE3] = {"any_cmd", arg, NULL};
  char  *const env[] = {NULL};

  /* Sanitize argument */

  /* Use execve() to execute any_cmd. */

  if (execve("/usr/bin/time", args, env) == -1) {
    /* Handle error */
  }
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [taintformatio]

Using a tainted value to write to an object using a formatted input or output function

Description

Rule Definition

Using a tainted value to write to an object using a formatted input or output function.

Polyspace Implementation

This checker checks for these issues:

- **Buffer overflow from incorrect string format specifier**.
- **Destination buffer overflow in string manipulation**.
- **Invalid use of standard library routine**.
- **Invalid use of standard library string routine**.
- **Tainted NULL or non-null-terminated string**.
- **Tainted string format specifier**.
- **Invalid use of standard library string routine**.
- **Use of dangerous standard function**.

Examples

Buffer overflow from incorrect string format specifier

Issue

Buffer overflow from incorrect string format specifier occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

Risk

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

Fix

Use a format specifier that is compatible with the memory buffer size.

Example - Memory Buffer Overflow

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf);
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

Correction — Use Smaller Precision in Format Specifier

One possible correction is to use a smaller precision in the format specifier.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

Destination buffer overflow in string manipulation

Issue

Destination buffer overflow in string manipulation occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

Example - Buffer Overflow in `sprintf` Use

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

Invalid use of standard library routine

Issue

This issue occurs when you use invalid arguments with a function from the standard library. This defect picks up errors related to other functions not covered by float, integer, memory, or string standard library routines.

Risk

Invalid arguments to a standard library function result in undefined behavior.

Fix

The fix depends on the root cause of the defect. For instance, the argument to a `printf` function can be `NULL` because a pointer was initialized with `NULL` and the initialization value was not overwritten along a specific execution path.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example – Calling `printf` Without a String

```
#include <stdio.h>
#include <stdlib.h>

void print_null(void) {

  printf(NULL);
}
```

The function `printf` takes only string input arguments or format specifiers. In this function, the input value is NULL, which is not a valid string.

Correction — Use Compatible Input Arguments

One possible correction is to change the input arguments to fit the requirements of the standard library routine. In this example, the input argument was changed to a character.

```
#include <stdio.h>

void print_null(void) {
    char zero_val = '0';
    printf((const char*)zero_val);
}
```

Invalid use of standard library string routine

Issue

Invalid use of standard library string routine occurs when a string library function is called with invalid arguments.

Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Invalid Use of Standard Library String Routine Error

```
 #include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
```

```
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

Tainted NULL or non-null-terminated string

Issue

This issue occurs when strings from nonsecure sources are used in string manipulation routines that implicitly dereference the string buffer, for instance, `strcpy` or `sprintf`.

The checker raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

Example – Getting String from Input Argument

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
```

```
  print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

Correction 1 — Validate the Data

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

int sanitize_str(char* s) {
  int res = 0;
  if (s && (strlen(s) > 0)) { // TAINTED_STRING only flagged here
    // - string is not null
    // - string has a positive and limited size
    // - TAINTED_STRING on strlen used as a firewall
    res = 1;
  }
  return res;
}

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
      strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  if (sanitize_str(userstr))
    strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}
```

Correction 2 — Validate the Data

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);
```

```
void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

Tainted string format specifier

Issue

This issue occurs when `printf`-style functions use a format specifier constructed from nonsecure sources.

Risk

If you use externally controlled elements to format a string, you can cause buffer overflow or data-representation problems. An attacker can use these string formatting elements to view the contents of a stack using `%x` or write to a stack using `%n`.

Fix

Pass a static string to format string functions. This fix ensures that an external actor cannot control the string.

Another possible fix is to allow only the expected number of arguments. If possible, use functions that do not support the vulnerable `%n` operator in format strings.

Example – Get Elements from User Input

```
#include "stdio.h"

void taintedstringformat(char* userstr) {
    printf(userstr);
}
```

This example prints the input argument `userstr`. The string is unknown. If it contains elements such as `%`, `printf` can interpret `userstr` as a string format instead of a string, causing your program to crash.

Correction — Print as String

One possible correction is to print `userstr` explicitly as a string so that there is no ambiguity.

```
#include "stdio.h"

void taintedstringformat(char* userstr) {
    printf("%.20s", userstr);
}
```

Use of dangerous standard function

Issue

The **Use of dangerous standard function** check highlights uses of functions that are inherently dangerous or potentially dangerous given certain circumstances. The following table lists possibly dangerous functions, the risks of using each function, and what function to use instead.

Dangerous Function	Risk Level	Safer Function
`gets`	Inherently dangerous — You cannot control the length of input from the console.	`fgets`
`cin`	Inherently dangerous — You cannot control the length of input from the console.	Avoid or prefaces calls to `cin` with `cin.width`.
`strcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`strncpy`
`stpcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`stpncpy`
`lstrcpy` or `StrCpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy`
`strcat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`strncat`, `strlcat`, or `strcat_s`
`lstrcat` or `StrCat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`StringCbCat`, `StringCchCat`, `strncay`, `strcat_s`, or `strlcat`
`wcpcpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`wcpncpy`

Dangerous Function	Risk Level	Safer Function
`wcscat`	Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur.	`wcsncat`, `wcslcat`, or `wcncat_s`
`wcscpy`	Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur.	`wcsncpy`
`sprintf`	Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur.	`snprintf`
`vsprintf`	Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur.	`vsnprintf`

Risk

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Using `sprintf`

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (sprintf(dst, "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

Correction — Use `snprintf` with Buffer Size

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [taintnoproto]

Using a tainted value as an argument to an unprototyped function pointer

Description

Rule Definition

Using a tainted value as an argument to an unprototyped function pointer.

Polyspace Implementation

This checker checks for **Call through non-prototyped function pointer**.

Examples

Call through non-prototyped function pointer

Issue

Call through non-prototyped function pointer detects a call to a function through a pointer without a prototype. A function prototype specifies the type and number of parameters.

Risk

Arguments passed to a function without a prototype might not match the number and type of parameters of the function definition, which can cause undefined behavior. If the parameters are restricted to a subset of their type domain, arguments from untrusted sources can trigger vulnerabilities in the called function.

Fix

Before calling the function through a pointer, provide a function prototype.

Example - Argument Does Not Match Parameter Restriction

```
#include <stdio.h>
#include <limits.h>
#define SIZE2 2

typedef void (*func_ptr)();
extern int getchar_wrapper(void);
extern void restricted_int_sink(int i);
/* Integer value restricted to
range [-1, 255] */
extern void restricted_float_sink(double i);
/* Double value restricted to > 0.0 */



func_ptr generic_callback[SIZE2] =
{
    (func_ptr)restricted_int_sink,
    (func_ptr)restricted_float_sink
};
```

```
void func(void)
{
    int ic;
    ic = getchar_wrapper();
    /* Wrong index used for generic_callback.
    Negative 'int' passed to restricted_float_sink. */
    (*generic_callback[1])(ic);
}
```

In this example, a call through `func_ptr` passes `ic` as an argument to function `generic_callback[1]`. The type of `ic` can have negative values, while the parameter of `generic_callback[1]` is restricted to float values greater than `0.0`. Typically, compilers and static analysis tools cannot perform type checking when you do not provide a pointer prototype.

Correction — Provide Prototype of Pointer to Function

Pass the argument `ic` to a function with a parameter of type `int`, by using a properly prototyped pointer.

```
#include <stdio.h>
#include <limits.h>
#define SIZE2 2

typedef void (*func_ptr_proto)(int);
extern int getchar_wrapper(void);
extern void restricted_int_sink(int i);
/* Integer value restricted to
range [-1, 255] */
extern void restricted_float_sink(double i);
/* Double value restricted to > 0.0 */

func_ptr_proto generic_callback[SIZE2] =
{
    (func_ptr_proto)restricted_int_sink,
    (func_ptr_proto)restricted_float_sink
};

void func(void)
{
    int ic;
    ic = getchar_wrapper();
    /* ic passed to function through
properly prototyped pointer. */
    (*generic_callback[0])(ic);
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [taintsink]

Tainted, potentially mutilated, or out-of-domain integer values are used in a restricted sink

Description

Rule Definition

Tainted, potentially mutilated, or out-of-domain integer values are used in a restricted sink.

Polyspace Implementation

This checker checks for these issues:

- **Tainted size of variable length array**.
- **Pointer dereference with tainted offset**.
- **Array access with tainted index**.

Examples

Tainted size of variable length array

Issue

Tainted size of variable length array detects variable length arrays (VLA) whose size is from an unsecure source.

Risk

If an attacker changed the size of your VLA to an unexpected value, it can cause your program to crash or behave unexpectedly.

If the size is non-positive, the behavior of the VLA is undefined. Your program does not perform as expected.

If the size is unbounded, the VLA can cause memory exhaustion or stack overflow.

Fix

Validate your VLA size to make sure that it is positive and less than a maximum value.

Example - Input Argument Used as Size of VLA

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedvlasize(int size) {

    int tabvla[size];
    int res = 0;
    for (int i=0 ; i<SIZE10 ; ++i) {
```

```
        tabvla[i] = i*i;
        res += tabvla[i];
    }
    return res;
}
```

In this example, a variable length array size is based on an input argument. Because this input argument value is not checked, the size may be negative or too large.

Correction — Check VLA Size

One possible correction is to check the size variable before creating the variable length array. This example checks if the size is larger than 10 and less than 100, before creating the VLA

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedvlasize(int size) {
    int res = 0;
    if (size>SIZE10 && size<SIZE100) {
        int tabvla[size];
        for (int i=0 ; i<SIZE10 ; ++i) {
            tabvla[i] = i*i;
            res += tabvla[i];
        }
    }
    return res;
}
```

Pointer dereference with tainted offset

Issue

Pointer dereference with tainted offset detects pointer dereferencing, either reading or writing, using an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

Fix

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

Example - Dereference Pointer Array

```
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(int i) {
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[i];
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `i`. The value of `i` could be outside the pointer range, causing an out-of-range error.

Correction — Check Index Before Dereference

One possible correction is to validate the value of the index. If the index is inside the valid range, continue with the pointer dereferencing.

```
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(int i) {
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (i>0 && i<SIZE10) {
            c = pint[i];
        }
        free(pint);
    }
    return c;
}
```

Array access with tainted index

Issue

Array access with tainted index detects reading or writing to an array by using a tainted index that has not been validated.

Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

Example - Use Index to Return Buffer Value

```
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    return tab[num];
}
```

In this example, the index `num` accesses the array `tab`. The function does not check to see if `num` is inside the range of `tab`.

Correction — Check Range Before Use

One possible correction is to check that `num` is in range before using it.

```
#define SIZE100 100
extern int tab[SIZE100];

int taintedarrayindex(int num) {
    if (num >= 0 && num < SIZE100) {
           return tab[num];
    } else {
        return -9999;
    }
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics
“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [taintstrcpy]

Tainted strings are passed to a string copying function

Description

Rule Definition

Tainted strings are passed to a string copying function.

Polyspace Implementation

This checker checks for **Tainted NULL or non-null-terminated string**.

Examples

Tainted NULL or non-null-terminated string

Issue

Tainted NULL or non-null-terminated string looks for strings from unsecure sources that are being used in string manipulation routines that implicitly dereference the string buffer. For example, `strcpy` or `sprintf`.

Tainted NULL or non-null-terminated string raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

Note If you reference a string using the form `ptr[i]`, `*ptr`, or pointer arithmetic, Bug Finder raises a **Use of tainted pointer** defect instead. The **Tainted NULL or non-null-terminated string** defect is raised only when the pointer is used as a string.

Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

Example - Getting String from Input Argument

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

Correction — Validate the Data

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

int sanitize_str(char* s) {
  int res = 0;
  if (s && (strlen(s) > 0)) { // TAINTED_STRING only flagged here
    // - string is not null
    // - string has a positive and limited size
    // - TAINTED_STRING on strlen used as a firewall
    res = 1;
  }
  return res;
}

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
      strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
```

```
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  if (sanitize_str(userstr))
    strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}
```

Correction — Validate the Data

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [uninitref]

Referencing uninitialized memory

Description

Rule Definition

Referencing uninitialized memory.

Polyspace Implementation

This checker checks for these issues:

- **Non-initialized pointer**.
- **Pointer to non-initialized value converted to const pointer**.
- **Non-initialized variable**.

Examples

Non-initialized pointer

Issue

Non-initialized pointer occurs when a pointer is not assigned an address before dereference.

Risk

Unless a pointer is explicitly assigned an address, it points to an unpredictable location.

Fix

The fix depends on the root cause of the defect. For instance, you assigned an address to the pointer but the assignment is unreachable.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below. It is a good practice to initialize a pointer to NULL when declaring the pointer.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Non-initialized pointer error

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;
```

```
    if (prev == NULL)
      {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
      }

    *pi = j;
    /* Defect: Writing to uninitialized pointer */

    return pi;
}
```

If `prev` is not `NULL`, the pointer `pi` is not assigned an address. However, `pi` is dereferenced on every execution paths, irrespective of whether `prev` is `NULL` or not.

Correction — Initialize Pointer on Every Execution Path

One possible correction is to assign an address to `pi` when `prev` is not `NULL`.

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
       {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
       }
    /* Fix: Initialize pi in branches of if statement  */
    else
        pi = prev;


    *pi = j;

    return pi;
}
```

Pointer to non-initialized value converted to const pointer

Issue

Pointer to non initialized value converted to const pointer occurs when a pointer to a constant (`const int*`, `const char*`, etc.) is assigned an address that does not yet contain a value.

Risk

A pointer to a constant stores a value that must not be changed later in the program. If you assign the address of a non-initialized variable to the pointer, it now points to an address with garbage values for the remainder of the program.

Fix

Initialize a variable before assigning its address to a pointer to a constant.

Example - Pointer to non initialized value converted to const pointer error

```
#include<stdio.h>

void Display_Parity()
 {
  int num,parity;
  const int* num_ptr = &num;
  /* Defect: Address &num does not store a value */

  printf("Enter a number\n:");
  scanf("%d",&num);

  parity=((*num_ptr)%2);
  if(parity==0)
    printf("The number is even.");
  else
    printf("The number is odd.");

 }
```

num_ptr is declared as a pointer to a constant. However the variable num does not contain a value when num_ptr is assigned the address &num.

Correction — Store Value in Address Before Assignment to Pointer

One possible correction is to obtain the value of num from the user before &num is assigned to num_ptr.

```
#include<stdio.h>

void Display_Parity()
 {
  int num,parity;
  const int* num_ptr;

  printf("Enter a number\n:");
  scanf("%d",&num);

 /* Fix: Assign &num to pointer after it receives a value */
  num_ptr=&num;
  parity=((*num_ptr)%2);
  if(parity==0)
    printf("The number is even.");
  else
    printf("The number is odd.");
 }
```

The scanf statement stores a value in &num. Once the value is stored, it is legitimate to assign &num to num_ptr.

Non-initialized variable

Issue

Non-initialized variable occurs when a variable is not initialized before its value is read.

Risk

Unless a variable is explicitly initialized, the variable value is unpredictable. You cannot rely on the variable having a specific value.

Fix

The fix depends on the root cause of the defect. For instance, you assigned a value to the variable but the assignment is unreachable or you assigned a value to the variable in one of two branches of a conditional statement. Fix the unreachable code or missing assignment.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Polyspace Bug Finder Results”.

See examples of fixes below. It is a good practice to initialize a variable at declaration.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Non-initialized variable error

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    int val;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
    /* Defect: val does not have a value if command is not 2 */
}
```

If `command` is not 2, the variable `val` is unassigned. In this case, the return value of function `get_sensor_value` is undetermined.

Correction — Initialize During Declaration

One possible correction is to initialize `val` during declaration so that the initialization is not bypassed on some execution paths.

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    /* Fix: Initialize val */
    int val=0;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
```

```
      }

    return val;
 }
```

`val` is assigned an initial value of 0. When `command` is not equal to 2, the function `get_sensor_value` returns this value.

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [usrfmt]

Including tainted or out-of-domain input in a format string

Description

Rule Definition

Including tainted or out-of-domain input in a format string.

Polyspace Implementation

This checker checks for **Format string specifiers and arguments mismatch**.

Examples

Format string specifiers and arguments mismatch

Issue

Format string specifiers and arguments mismatch occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
  const char *message = "License not available";
  int err_number = ;-4
  printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See “Address Polyspace Results Through Bug Fixes or Justifications”.

Example - Printing a Float

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", fst);
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [xfilepos]

Using a value for fsetpos other than a value returned from fgetpos

Description

Rule Definition

Using a value for fsetpos other than a value returned from fgetpos.

Polyspace Implementation

This checker checks for **Invalid file position**.

Examples

Invalid file position

Issue

Invalid file position occurs when the file position argument of `fsetpos()` uses a value that is not obtained from `fgetpos()`.

Risk

The function `fgetpos(FILE *stream, fpos_t *pos)` gets the current file position of the stream. When you use any other value as the file position argument of `fsetpos(FILE *stream, const fpos_t *pos)`, you might access an unintended location in the stream.

Fix

Use the value returned from a successful call to `fgetpos()` as the file position argument of `fsetpos()`.

Example - `memset()` Sets File Position Argument

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


FILE *func(FILE *file)
{
    fpos_t offset;
    if (file == NULL)
    {
        /* Handle error */
    }
    /* Store initial position in variable 'offset' */
    (void)memset(&offset, 0, sizeof(offset));

    /* Read data from file */

    /* Return to the initial position. offset was not
    returned from a call to fgetpos()    */
```

```
    if (fsetpos(file, &offset) != 0)
    {
        /* Handle error */
    }
    return file;
}
```

In this example, `fsetpos()` uses `offset` as its file position argument. However, the value of `offset` is set by `memset()`. The preceding code might access the wrong location in the stream.

Correction — Use a File Position Returned From `fgetpos()`

Call `fgetpos()`, and if it returns successfully, use the position argument in your call to `fsetpos()`.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

FILE *func(FILE *file)
{
    fpos_t offset;
    if (file == NULL)
    {
        /* Handle error */
    }
    /* Store initial position in variable 'offset'
    using fgetpos() */
    if (fgetpos(file, &offset) != 0)
    {
        /* Handle error */
    }

    /* Read data from file */

    /* Back to the initial position */
    if (fsetpos(file, &offset) != 0)
    {
        /* Handle error */
    }
    return file;
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

ISO/IEC TS 17961 [xfree]

Reallocating or freeing memory that was not dynamically allocated

Description

Rule Definition

Reallocating or freeing memory that was not dynamically allocated.

Polyspace Implementation

This checker checks for **Invalid free of pointer**.

Examples

Invalid free of pointer

Issue

Invalid free of pointer occurs when a block of memory released using the `free` function was not previously allocated using `malloc`, `calloc`, or `realloc`.

Risk

The `free` function releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.

The issue can highlight coding errors. For instance, you perhaps wanted to use the `free` function or a previous `malloc` function on a different pointer.

Fix

In most cases, you can fix the issue by removing the `free` statement. If the pointer is not allocated memory from the heap with `malloc` or `calloc`, you do not need to free the pointer. You can simply reuse the pointer as required.

If the issue highlights a coding error such as use of `free` or `malloc` on the wrong pointer, correct the error.

If the issue occurs because you use the `free` function to free memory allocated with the `new` operator, replace the `free` function with the `delete` operator.

Example - Invalid Free of Pointer Error

```
#include <stdlib.h>

void Assign_Ones(void)
{
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;

  free(p);
```

```
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer `p` is deallocated using the `free` function. However, `p` points to a memory location that was not dynamically allocated.

Correction — Remove Pointer Deallocation

If the number of elements of the array `p` is known at compile time, one possible correction is to remove the deallocation of the pointer `p`.

```
#include <stdlib.h>

void Assign_Ones(void)
 {
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;
  /* Fix: Remove deallocation of p */
 }
```

Correction — Introduce Pointer Allocation

If the number of elements of the array `p` is not known at compile time, one possible correction is to dynamically allocate memory to the array `p`.

```
#include <stdlib.h>

void Assign_Ones(int num)
{
  int *p;
  /* Fix: Allocate memory dynamically to p */
  p=(int*) calloc(10,sizeof(int));
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
}
```

Check Information

Decidability: Undecidable

See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

Topics

“Check for Coding Standard Violations”

Introduced in R2019a

Custom Coding Rules

Group 1: Files

The custom rules 1.*x* in Polyspace enforce naming conventions for files and folders. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
1.1	All source file names must follow the specified pattern.	Only the base name is checked. A source file is a file that is not included.
1.2	All source folder names must follow the specified pattern.	Only the folder name is checked. A source file is a file that is not included.
1.3	All include file names must follow the specified pattern.	Only the base name is checked. An include file is a file that is included.
1.4	All include folder names must follow the specified pattern.	Only the folder name is checked. An include file is a file that is included.

Group 2: Preprocessing

The custom rules 2.*x* in Polyspace enforce naming conventions for macros. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
2.1	All macros must follow the specified pattern.	Macro names are checked before preprocessing.
2.2	All macro parameters must follow the specified pattern.	Macro parameters are checked before preprocessing.

Group 3: Type definitions

The custom rules 3.*x* in Polyspace enforce naming conventions for fundamental data types. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
3.1	All integer types must follow the specified pattern.	Applies to integer types specified by `typedef` statements. Does not apply to enumeration types. For example: `typedef signed int int32_t;`
3.2	All float types must follow the specified pattern.	Applies to float types specified by `typedef` statements. For example: `typedef float f32_t;`
3.3	All pointer types must follow the specified pattern.	Applies to pointer types specified by `typedef` statements. For example: `typedef int* p_int;`
3.4	All array types must follow the specified pattern.	Applies to array types specified by `typedef` statements. For example: `typedef int[3] a_int_3;`
3.5	All function pointer types must follow the specified pattern.	Applies to function pointer types specified by `typedef` statements. For example: `typedef void (*pf_callback) (int);`

Group 4: Structures

The custom rules 4.*x* in Polyspace enforce naming conventions for structured data types. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
4.1	All `struct` tags must follow the specified pattern.	
4.2	All `struct` types must follow the specified pattern.	`struct` types are aliases for previously defined structures (defined with the `typedef` or `using` keyword).
4.3	All `struct` fields must follow the specified pattern.	
4.4	All `struct` bit fields must follow the specified pattern.	

Group 5: Classes (C++)

The custom rules 5.*x* in Polyspace enforce naming conventions for classes and class members. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
5.1	All class names must follow the specified pattern.	
5.2	All class types must follow the specified pattern.	Class types are aliases for previously defined classes (defined with the `typedef` or `using` keyword).
5.3	All data members must follow the specified pattern.	
5.4	All function members must follow the specified pattern.	
5.5	All static data members must follow the specified pattern.	
5.6	All static function members must follow the specified pattern.	
5.7	All bitfield members must follow the specified pattern.	

Group 6: Enumerations

The custom rules 6.*x* in Polyspace enforce naming conventions for enumerations. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
6.1	All enumeration tags must follow the specified pattern.	
6.2	All enumeration types must follow the specified pattern.	Enumeration types are aliases for previously defined enumerations (defined with the `typedef` or `using` keyword).
6.3	All enumeration constants must follow the specified pattern.	

Group 7: Functions

The custom rules 7.*x* in Polyspace enforce naming conventions for functions and function parameters. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
7.1	All global functions must follow the specified pattern.	A global function is a function with external linkage.
7.2	All static functions must follow the specified pattern.	A static function is a function with internal linkage.
7.3	All function parameters must follow the specified pattern.	In C++, applies to non-member functions.

Group 8: Constants

The custom rules 8.*x* in Polyspace enforce naming conventions for constants. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
8.1	All global constants must follow the specified pattern.	A global constant is a constant with external linkage.
8.2	All static constants must follow the specified pattern.	A static constant is a constant with internal linkage.
8.3	All local constants must follow the specified pattern.	A local constant is a constant without linkage.
8.4	All static local constants must follow the specified pattern.	A static local constant is a constant declared static in a function.

Group 9: Variables

The custom rules 9.*x* in Polyspace enforce naming conventions for variables. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
9.1	All global variables must follow the specified pattern.	A global variable is a variable with external linkage.
9.2	All static variables must follow the specified pattern.	A static variable is a variable with internal linkage.
9.3	All local variables must follow the specified pattern.	A local variable is a variable without linkage.
9.4	All static local variables must follow the specified pattern.	A static local variable is a variable declared static in a function.

Group 10: Name spaces (C++)

The custom rules 10.*x* in Polyspace enforce naming conventions for namespaces. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied
10.1	All names spaces must follow the specified pattern.

Group 11: Class templates (C++)

The custom rules 11.*x* in Polyspace enforce naming conventions for class templates. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
11.1	All class templates must follow the specified pattern.	
11.2	All class template parameters must follow the specified pattern.	

Group 12: Function templates (C++)

The custom rules 12.*x* in Polyspace enforce naming conventions for function templates. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
12.1	All function templates must follow the specified pattern.	Applies to non-member functions.
12.2	All function template parameters must follow the specified pattern.	Applies to non-member functions.
12.3	All function template members must follow the specified pattern.	

Group 20: Style

The custom rules 20.*x* in Polyspace enforce coding style conventions such as number of characters per line. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
20.1	Source line length must not exceed specified number of characters.	When configuring the checker, specify: • A number for the character limit. Use the **Pattern** column on the configuration or the `pattern=` line in the custom rules text file. • A violation message such as: `Line exceeds` *`n`* `characters.` Use the **Convention** column on the configuration or the `convention=` line in the custom rules xml file.

Code Metrics

Comment Density

Ratio of number of comments to number of statements

Description

The metric specifies the ratio of comments to statements expressed as a percentage.

Based on HIS specifications:

- Multi-line comments count as one comment.

  For instance, the following constitutes one comment:

  ```
  // This function implements
  // regular maintenance on an internal database
  ```

- Comments that start with the source code line do not count as comments.

  For instance, this comment does not count as a comment for the metric but counts as a statement instead:

  ```
  remove(i); // Remove employee record
  ```

- A statement typically ends with a semi-colon with some exceptions. Exceptions include semi-colons in `for` loops or structure field declarations.

  For instance, the initialization, condition and increment within parentheses in a `for` loop is counted as one statement. The following counts as one statement:

  ```
  for(i=0; i <100; i++)
  ```

  If you also declare the loop counter at initialization, it counts as two statements.

The recommended lower limit for this metric is 20. For better readability of your code, try to place at least one comment for every five statements.

To enforce limits on metrics, see “Compute Code Complexity Metrics”.

Examples

Comment Density Calculation

```
struct record {
    char name[40];
    long double salary;
    int isEmployed;
};

struct record dataBase[100];

struct record fetch(void);
void remove(int);

void maintenanceRoutines() {
```

```
// This function implements
// regular maintenance on an internal database
    int i;
    struct record tempRecord;

    for(i=0; i <100; i++) {
        tempRecord = fetch(); // This function fetches a record
        // from the database
        if(tempRecord.isEmployed == 0)
            remove(i);         // Remove employee record
        //from the database
    }
}
```

In this example, the comment density is 38. The calculation is done as follows:

Code	Running Total of Comments	Running Total of Statements

`struct record {`
`  char name[40];`
`  long double salary;`
`  int isEmployed;`
`};`
	0	1

`struct record dataBase[100];`
`struct record fetch(void);`
`void remove(int);`
	0	4

`void maintenanceRoutines() {`	0	4
`// This function implements` `// regular maintenance on an internal database`	1	4
`int i;` `struct record tempRecord;`	1	6
`for(i=0; i <100; i++) {`	1	6

`tempRecord = fetch(); // This`
`function fetches a record`
`// from the database`
	2	7

`if(tempRecord.isEmployed == 0)`
`remove(i);`
`// Remove employee record`
`//from the database`
`}`
`}`
	3	8

There are 3 comments and 8 statements. The comment density is 3/8*100 = 38.

Metric Information

Group: File
Acronym: COMF
HIS Metric: Yes

See Also

Calculate code metrics (-code-metrics)

Cyclomatic Complexity

Number of linearly independent paths in function body

Description

This metric calculates the number of decision points in a function and adds one to the total. A decision point is a statement that causes your program to branch into two paths.

The recommended upper limit for this metric is 10. If the cyclomatic complexity is high, the code is both difficult to read and can cause more orange checks. Therefore, try to limit the value of this metric.

To enforce limits on metrics, see “Compute Code Complexity Metrics”.

Computation Details

The metric calculation uses the following rules to identify decision points:

- An `if` statement is one decision point.
- The statements `for` and `while` count as one decision point, even when no condition is evaluated, for example, in infinite loops.
- Boolean combinations `(&&, ||)` do not count as decision points.
- `case` statements do not count as decision points unless they are followed by a `break` statement. For instance, this code has a cyclomatic complexity of two:

```
switch(num) {
        case 0:
        case 1:
        case 2:
            break;
        case 3:
        case 4:
   }
```

- The calculation is done after preprocessing:
  - Macros are expanded.
  - Conditional compilation is applied. The blocks hidden by preprocessing directives are ignored.

Examples

Function with Nested `if` Statements

```
int foo(int x,int y)
{
    int flag;
    if (x <= 0)
        /* Decision point 1*/
        flag = 1;
    else
```

```
    {
        if (x < y )
            /* Decision point 2*/
            flag = 1;
        else if (x==y)
            /* Decision point 3*/
            flag = 0;
        else
            flag = -1;
    }
    return flag;
}
```

In this example, the cyclomatic complexity of `foo` is 4.

Function with ? Operator

```
int foo (int x, int y) {
    if((x <0) ||(y < 0))
        /* Decision point 1*/
        return 0;
    else
        return (x > y ? x: y);
        /* Decision point 2*/
}
```

In this example, the cyclomatic complexity of `foo` is 3. The `?` operator is the second decision point.

Function with `switch` Statement

```
#include <stdio.h>

int foo(int x,int y, int ch)
{
    int val = 0;
    switch(ch) {
    case 1:
        /* Decision point 1*/
        val = x + y;
        break;
    case 2:
        /* Decision point 2*/
        val = x - y;
        break;
    default:
        printf("Invalid choice.");
    }
    return val;
}
```

In this example, the cyclomatic complexity of `foo` is 3.

Function with Nesting of Different Control-Flow Statements

```
int foo(int x,int y, int bound)
{
    int count = 0;
    if (x <= y)
```

```
        /* Decision point 1*/
        count = 1;
    else
        while(x>y) {
            /* Decision point 2*/
            x--;
            if(count< bound) {
                /* Decision point 3*/
                count++;
            }
        }
    return count;
}
```

In this example, the cyclomatic complexity of `foo` is 4.

Metric Information

Group: Function
Acronym: VG
HIS Metric: Yes

See Also

`Calculate code metrics (-code-metrics)`

Estimated Function Coupling

Measure of complexity between levels of call tree

Description

This metric provides an approximate measure of complexity between different levels of the call tree. The metric is defined as:

number of call occurrences – number of function definitions + 1

If there are more function definitions than function calls, the estimated function coupling result is negative.

This metric:

- Counts function calls and function definitions in the current file only.

  It does not count function definitions in a header file included in the current file.
- Treats `static` and `inline` functions like any other function.

Examples

Same Function Called Multiple Times

```
void checkBounds(int *);
int getUnboundedValue();

int getBoundedValue(void) {
    int num = getUnboundedValue();
    checkBounds(&num);
    return num;
}

void main() {
    int input1=getBoundedValue(), input2= getBoundedValue(), prod;
    prod = input1 * input2;
    checkBounds(&prod);
}
```

In this example, there are:

- 5 call occurrences. Both `getBoundedValue` and `checkBounds` are called twice and `getUnboundedValue` is called once.
- 2 function definitions. `main` and `getBoundedValue` are defined.

Therefore, the Estimated function coupling is `5 - 2 + 1 = 4`.

Negative Estimated Function Coupling

```
int foobar(int a, int b){
    return a+b;
}
```

```
int bar(int b){
    return b+2;
}

int foo(int a){
    return a<<2;
}

int main(int x){
    foobar(x,x+2);
    return 0;
}
```

This example shows how you can get a negative estimated function coupling result. In this example, you see:

- `1` function call in `main`.
- `4` defined functions: `foobar`, `bar`, `foo`, and `main`.

Therefore, the estimated function coupling is `1 - 4 + 1 = -2`.

Metric Information

Group: File
Acronym: `FCO`
HIS Metric: No

See Also

`Number of Call Occurrences` | `Calculate code metrics (-code-metrics)`

Higher Estimate of Local Variable Size

Total size of all local variables in function

Description

This metric provides a conservative estimate of the total size of local variables in a function. The metric is the sum of the following sizes in bytes:

- Size of function return value
- Sizes of function parameters
- Sizes of local variables
- Additional padding introduced for memory alignment

Your actual stack usage due to local variables can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. For instance, compilers store the address to which the execution returns following the function call. When computing this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. When computing this metric, Polyspace does not consider this hidden memory usage.

However, the metric provides a reasonable estimate of the stack usage due to local variables.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric also takes into account `#pragma pack` directives in your code.

Examples

All Variables of Same Type

```
int flag();

int func(int param) {
  int var_1;
  int var_2;
  if (flag()) {
      int var_3;
      int var_4;
    } else {
      int var_5;
    }
}
```

In this example, assuming 4 bytes for `int`, the higher estimate of local variable size is 28. The breakup of the size is shown in this table.

Variable	Size (in Bytes)	Running Total
Return value	4	4

Variable	Size (in Bytes)	Running Total
Parameter `param`	4	8
Local variables `var_1` and `var_2`	`4+4=8`	16
Local variables defined in the `if` condition	`(4+4)+4=12` The size of variables in the first branch is eight bytes. The size in the second branch is four bytes. The sum of the two branches is 12 bytes.	28

No padding is introduced for memory alignment because all the variables involved have the same type.

Variables of Different Types

```
char func(char param) {
  int var_1;
  char var_2;
  double var_3;
}
```

In this example, assuming one byte for `char`, four bytes for `int` and eight bytes for `double` and four bytes for alignment, the higher estimate of local variable size is 20. The alignment is usually the word size on your platform. In your Polyspace project, you specify the alignment through your target processor. For more information, see the Alignment column in `Target processor type (-target)`.

The breakup of the size is shown in this table.

Variable	Size (in Bytes)	Running Total
Return value	1	1
Additional padding introduced before `param` is stored	0 No memory alignment is required because the next variable `param` has the same size.	1
Parameter `param`	1	2
Additional padding introduced before `var_1` is stored	2 Memory must be aligned using padding because the next variable `var_1` requires four bytes. The storage must start from a memory address at a multiple of four.	4
`var_1`	4	8

Variable	Size (in Bytes)	Running Total
Additional padding introduced before `var_2` is stored	0 No memory alignment is required because the next variable `var_2` has smaller size.	8
`var_2`	1	9
Additional padding introduced before `var_3` is stored	3 Memory must be aligned using padding because the next variable `var_3` has eight bytes. The storage must start from a memory address at a multiple of the alignment, four bytes.	12
`var_3`	8	20

The rules for the amount of padding are:

- If the next variable stored has the same or smaller size, no padding is required.
- If the next variable has a greater size:
  - If the variable size is the same as or less than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of its size.
  - If the variable size is greater than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of the alignment.

C++ Methods and Objects

```
class MySimpleClass {
  public:
    MySimpleClass() {};
    MySimpleClass(int) {};
    ~MySimpleClass() {};
};

int main() {
  MySimpleClass c;
  return 0;
}
```

In this example, the estimated local variable sizes are:

- Constructor `MySimpleClass::MySimpleClass()`: Four bytes.

  The size comes from the `this` pointer, which is an implicit argument to the constructor. You specify the pointer size using the option `Target processor type (-target)`.
- Constructor `MySimpleClass::MySimpleClass(int)`: Eight bytes.

  The size comes from the `this` pointer and the `int` argument.
- Destructor `MySimpleClass::~MySimpleClass()`: Four bytes.

  The size comes from the `this` pointer.

- `main()`: Five bytes.

  The size comes from the `int` return value and the size of object `c`. The minimum size of an object is the alignment that you specify using the option `Target processor type (-target)`.

C++ Functions with Object Arguments

```
class MyClass {
  public:
    MyClass() {};
    MyClass(int) {};
    ~MyClass() {};
  private:
    int i[10];
};
void func1(const MyClass& c) {
}

void func2() {
  func1(4);
}
```

In this example, the estimated local variable size for `func2()` is 40 bytes. When `func2()` calls `func1()`, a temporary object of the class `MyClass` is created. The object has ten `int` variables, each with a size of four bytes.

Metric Information

Group: Function
Acronym: `LOCAL_VARS_MAX`
HIS Metric: No

See Also

`Lower Estimate of Local Variable Size` | `Calculate code metrics (-code-metrics)`

Introduced in R2016b

Language Scope

Language scope

Description

This metric measures the cost of maintaining or changing a function. It is calculated as:

```
(N1 + N2)/(n1 + n2)
```

Here:

- `N1` is the number of occurrences of operators.

  Other than identifiers (variable or function names) and literal constants, everything else counts as operators.
- `N2` is the number of occurrences of operands.
- `n1` is the number of distinct operators.
- `n2` is the number of distinct operands.

  The metric considers a literal constant with a suffix as different from the constant without the suffix. For instance, 0 and `0U` are considered different.

Tip To find `N1 + N2`, count the total number of tokens. To find `n1 + n2`, count the number of unique tokens.

The recommended upper limit for this metric is 4. For lower maintenance cost for a function, try to enforce an upper limit on this metric. For instance, if the same operand occurs many times, to change the operand name, you have to make many substitutions.

To enforce limits on metrics, see “Compute Code Complexity Metrics”.

Examples

Language Scope Calculation

```
int f(int i)
{
    if (i == 1)
        return i;
    else
        return i * g(i-1);
}
```

In this example:

- `N1` = 19.
- `N2` = 9.
- `n1` = 12.

The distinct operators are `int`, `(`, `)`, `{`, `if`, `==`, `return`, `else`, `*`, `-`, `;`, `}`.

- `n2` = 4.

  The distinct operands are `f`, `i`, `1` and `g`.

The language scope of `f` is (17 + 9) / (12 + 4) = 1.8.

C++ Namespaces in Language Scope Calculation

```
namespace std {
  int func2() {
    return 123;
  }
};

namespace my_namespace {
  using namespace std;
  int func1(int a, int b) {
    return func2();
  }
};
```

In this example, the namespace `std` is implicitly associated with `func2`. The language scope computation treats `func2()` as `std::func2()`. Likewise, the computation treats `func1()` as `my_namespace::func1()`.

For instance, the language scope value for `func1` is 1.3. To break down this calculation:

- `N1 + N2` = 20.
- `n1 + n2` = 15.

  The distinct operators are `int`, `::`, `(`, comma, `)`, `{`, `return`, `;`, and `}`.

  The distinct operands are `my_namespace`, `func1`, `a`, `b`, `std`, and `func2`.

Metric Information

Group: Function
Acronym: `VOCF`
HIS Metric: Yes

See Also

`Calculate code metrics (-code-metrics)`

Lower Estimate of Local Variable Size

Total size of local variables in function taking nested scopes into account

Description

This metric provides an optimistic estimate of the total size of local variables in a function. The metric is the sum of the following sizes in bytes:

- Size of function return value
- Sizes of function parameters
- Sizes of local variables

  Suppose that the function has variable definitions in nested scopes as follows:

  ```
  type func (type param_1, ...) {

    {
      /* Scope 1 */
      type var_1, ...;
    }
    {
      /* Scope 2 */
      type var_2, ...;
    }
  }
  ```

  The software computes the total variable size in each scope and uses whichever total is greatest. For instance, if a conditional statement has variable definitions, the software computes the total variable size in each branch, and then uses whichever total is greatest. If a nested scope itself has further nested scopes, the same process is repeated for the inner scopes.

  A variable defined in a nested scope is not visible outside the scope. Therefore, some compilers reuse stack space for variables defined in separate scopes. This metric provides a more accurate estimate of stack usage for such compilers. Otherwise, use the metric `Higher Estimate of Local Variable Size`. This metric adds the size of all local variables, whether or not they are defined in nested scopes.
- Additional padding introduced for memory alignment

Your actual stack usage due to local variables can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. When computing this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. For instance, compilers store the address to which the execution returns following the function call. When computing this metric, Polyspace does not consider this hidden memory usage.

However, the metric provides a reasonable estimate of the stack usage due to local variables.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric also takes into account `#pragma pack` directives in your code.

Examples

All Variables of Same Type

```
int flag();

int func(int param) {
  int var_1;
  int var_2;
  if (flag()) {
      int var_3;
      int var_4;
    } else {
      int var_5;
    }
}
```

In this example, assuming four bytes for `int`, the lower estimate of local variable size is 24. The breakup of the metric is shown in this table.

Variable	Size (in Bytes)	Running Total
Return value	4	4
Parameter `param`	4	8
Local variables `var_1` and `var_2`	`4+4=8`	16
Local variables defined in the `if` condition	`max(4+4,4)= 8` The size of variables in the first branch is eight bytes. The size in the second branch is four bytes. The maximum of the two branches is eight bytes.	24

No padding is introduced for memory alignment because all the variables involved have the same type.

Variables of Different Types

```
char func(char param) {
  int var_1;
  char var_2;
  double var_3;
}
```

In this example, assuming one byte for `char`, four bytes for `int`, eight bytes for `double` and four bytes for alignment, the lower estimate of local variable size is 20. The alignment is usually the word size on your platform. In your Polyspace project, you specify the alignment through your target processor. For more information, see the Alignment column in `Target processor type (-target)`.

The breakup of the size is shown in this table.

Variable	Size (in Bytes)	Running Total
Return value	1	1
Additional padding introduced before `param` is stored	0 No memory alignment is required because the next variable `param` has the same size.	1
Parameter `param`	1	2
Additional padding introduced before `var_1` is stored	2 Memory must be aligned using padding because the next variable `var_1` requires four bytes. The storage must start from a memory address at a multiple of four.	4
`var_1`	4	8
Additional padding introduced before `var_2` is stored	0 No memory alignment is required because the next variable `var_2` has smaller size.	8
`var_2`	1	9
Additional padding introduced before `var_3` is stored	3 Memory must be aligned using padding because the next variable `var_3` requires eight bytes. The storage must start from a memory address at a multiple of the alignment, four bytes.	12
`var_3`	8	20

The rules for the amount of padding are:

- If the next variable stored has the same or smaller size, no padding is required.
- If the next variable has a greater size:
  - If the variable size is the same as or less than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of its size.
  - If the variable size is greater than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of the alignment.

C++ Methods and Objects

```
class MySimpleClass {
  public:
    MySimpleClass() {};
```

```
    MySimpleClass(int) {};
    ~MySimpleClass() {};
};

int main() {
  MySimpleClass c;
  return 0;
}
```

In this example, the estimated local variable sizes are:

- Constructor `MySimpleClass::MySimpleClass()`: Four bytes.

  The size comes from the `this` pointer, which is an implicit argument to the constructor. You specify the pointer size using the option `Target processor type (-target)`.
- Constructor `MySimpleClass::MySimpleClass(int)`: Eight bytes.

  The size comes from the `this` pointer and the `int` argument.
- Destructor `MySimpleClass::~MySimpleClass()`: Four bytes.

  The size comes from the `this` pointer.
- `main()`: Five bytes.

  The size comes from the `int` return value and the size of object `c`. The minimum size of an object is the alignment that you specify using the option `Target processor type (-target)`.

C++ Functions with Object Arguments

```
class MyClass {
  public:
    MyClass() {};
    MyClass(int) {};
    ~MyClass() {};
  private:
    int i[10];
};
void func1(const MyClass& c) {
}

void func2() {
  func1(4);
}
```

In this example, the estimated local variable size for `func2()` is 40 bytes. When `func2()` calls `func1()`, a temporary object of the class `MyClass` is created. The object has ten `int` variables, each with a size of four bytes.

Metric Information

Group: Function
Acronym: `LOCAL_VARS_MIN`
HIS Metric: No

See Also

`Higher Estimate of Local Variable Size` | `Calculate code metrics (-code-metrics)`

Introduced in R2016b

Maximum Stack Usage

Total size of local variables in function plus maximum stack usage from callees

Description

This metric is reported in a Code Prover analysis only.

This metric provides a conservative estimate of the stack usage by a function. The metric is the sum of these sizes in bytes:

-
- Maximum value from the stack usages of the function callees. The computation uses the maximum stack usage of each callee.

  For instance, in this example, the maximum stack usage of `func` is the same as the maximum stack usage of `func1` or `func2`, *whichever is greater*.

  ```
  void func(void) {
      func1();
      func2();
  }
  ```

  If the function calls are in different branches of a conditional statement, this metric considers the branch with the greatest stack usage.

  The analysis does the stack size estimation later on when it has resolved which function calls actually occur. For instance, if a function call occurs in unreachable code, the stack size does not take the call into account. The analysis can also take into account calls through function pointers.

Your actual stack usage can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. When estimating this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. For instance, compilers store the address to which the execution returns following the function call. When estimating this metric, Polyspace does not consider this hidden memory usage.

However, the metric provides a reasonable estimate of the stack usage.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric takes into account `#pragma pack` directives in your code.

Examples

Function with One Callee

```
double func(int);
double func2(int);

double func(int status) {
```

```
    double res = func2(status);
    return res;
}

double func2(int status) {
    double res;
    if(status == 0) {
      int temp;
      res = 0.0;
    }
    else {
      double temp;
      res = 1.0;
    }
    return res;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func2`: 32 bytes

  This value includes the sizes of its parameter (4 bytes), local variable `res` (8 bytes), local variable `temp` counted twice (4+8=12 bytes), and return value (8 bytes).

  The metric does not take into account that the first `temp` is no longer live when the second `temp` is defined.
- `func`: 52 bytes

  This value includes the sizes of its parameter, local variable `res`, and return value, a total of 20 bytes. This value includes the 32 bytes of maximum stack usage by its callee, `func2`.

Function with Multiple Callees

```
void func1(int);
void func2(void);


void func(int status) {
    func1(status);
    func2();
}

void func1(int status) {
    if(status == 0) {
      int val;
    }
    else {
      double val2;
    }
}

void func2(void) {
    double val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func1`: 16 bytes

  This value includes the sizes of its parameter (4 bytes) and local variable `temp` counted twice (4+8=12 bytes).
- `func2`: 8 bytes
- `func`: 20 bytes

  This value includes the sizes of its parameter (4 bytes) and the maximum of stack usages of `func1` and `func2` (16 bytes).

Function with Multiple Callees in Different Branches

```
void func1(void);
void func2(void);


void func(int status) {
    if(status==0)
        func1();
    else
        func2();
}

void func1(void) {
    double val;
}

void func2(void) {
    int val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func1`: 8 bytes
- `func2`: 4 bytes
- `func`: 12 bytes

  This value includes the sizes of its parameter (4 bytes) and the maximum stack usage from the two branches (8 bytes).

Functions with Variable Number of Parameters (Variadic Functions)

```
#include <stdarg.h>

void fun_vararg(int x, ...) {
  va_list ap;
  va_start(ap, x);
  int i;
  for (i=0; i<x; i++) {
    int j = va_arg(ap, int);
  }
  va_end(ap);
}
```

```
void call_fun_vararg1(void) {
  long long int l = 0;
  fun_vararg(3, 4, 5, 6, l);
}


void call_fun_vararg2(void) {
  fun_vararg(1,0);
}
```

In this function, `fun_vararg` is a function with variable number of parameters. The maximum stack usage of `fun_vararg` takes into account the call to `fun_vararg` with the maximum number of arguments. The call with the maximum number of arguments is the call in `call_fun_vararg1` with five arguments (one for the fixed parameter and four for the variable parameters). The maximum stack usages are:

- `fun_vararg`: 36 bytes.

  This value takes into account:

  - The size of the fixed parameter `x` (4 bytes).
  - The sizes of the variable parameters from the call with the maximum number of parameters. In that call, there are four variable arguments: three `int` and one `long long int` variable (3 times 4 + 1 times 8 = 20 bytes).
  - The sizes of the local variables `i`, `j` and `ap` (12 bytes). The size of the `va_list` variable uses the pointer size defined in the target (in this case, 4 bytes).
- `call_fun_vararg1`: 44 bytes.

  This value takes into account:

  - The stack size usage of `fun_vararg` with five arguments (36 bytes).
  - The size of local variable `l` (8 bytes).
- `call_fun_vararg2`: 20 bytes.

  Since `call_fun_vararg2` has no local variables, this value is the same as the stack size usage of `fun_vararg` with two arguments (20 bytes, of which 12 bytes are for the local variables and 8 bytes are for the two parameters of `fun_vararg`).

Metric Information

Group: Function
Acronym: `MAX_STACK`
HIS Metric: No

See Also

`Calculate code metrics (-code-metrics)`

Introduced in R2017b

Minimum Stack Usage

Total size of local variables in function taking nested scopes into account plus maximum stack usage from callees

Description

This metric is reported in a Code Prover analysis only.

This metric provides an optimistic estimate of the stack usage by a function. Unlike the metric , this metric takes nested scopes into account. For instance, if variables are defined in two mutually exclusive branches of a conditional statement, the metric considers that the stack space allocated to the variables in one branch can be reused in the other branch.

The metric is the sum of these sizes in bytes:

- .
- Maximum value from the stack usages of the function callees. The computation uses the minimum stack usage of each callee.

  For instance, in this example, the minimum stack usage of `func` is the same as the minimum stack usage of `func1` or `func2`, *whichever is greater*.

  ```
  void func(void) {
      func1();
      func2();
  }
  ```

  If the function calls are in different branches of a conditional statement, this metric considers the branch with the least stack usage.

  The analysis does the stack size estimation later on when it has resolved which function calls actually occur. For instance, if a function call occurs in unreachable code, the stack size does not take the call into account. The analysis can also take into account calls through function pointers.

Your actual stack usage can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. When estimating this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. For instance, compilers store the address to which the execution returns following the function call. When estimating this metric, Polyspace does not consider this hidden memory usage.

However, the metric provides a reasonable estimate of the stack usage.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric takes into account `#pragma pack` directives in your code.

Examples

Function with One Callee

```
double func2(int);

double func(int status) {
    double res = func2(status);
    return res;
}

double func2(int status) {
    double res;
    if(status == 0) {
      int temp;
      res = 0.0;
    }
    else {
      double temp;
      res = 1.0;
    }
    return res;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func2`: 28 bytes

  This value includes the sizes of its parameter (4 bytes), local variable `res` (8 bytes), one of the two local variables `temp` (8 bytes), and return value (8 bytes).

  The metric takes into account that the first `temp` is no longer live when the second `temp` is defined. It uses the variable `temp` with data type `double` because its size is greater.

- `func`: 48 bytes

  This value includes the sizes of its parameter, local variable `res`, and return value, a total of 20 bytes. This value includes the 28 bytes of minimum stack usage by its callee, `func2`.

Function with Multiple Callees

```
void func1(int);
void func2(void);


void func(int status) {
    func1(status);
    func2();
}

void func1(int status) {
    if(status == 0) {
      int val;
    }
    else {
      double val2;
    }
```

```
}

void func2(void) {
    double val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func1`: 12 bytes

  This value includes the sizes of its parameter (4 bytes) and one of the two local variables `temp` (8 bytes). The metric takes into account that the first `temp` is no longer live when the second `temp` is defined.
- `func2`: 8 bytes
- `func`: 16 bytes

  This value includes the sizes of its parameter (4 bytes) and the maximum of stack usages of `func1` and `func2` (12 bytes).

Function with Multiple Callees in Different Branches

```
void func1(void);
void func2(void);


void func(int status) {
    if(status==0)
        func1();
    else
        func2();
}

void func1(void) {
    double val;
}

void func2(void) {
    int val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func1`: 8 bytes
- `func2`: 4 bytes
- `func`: 8 bytes

  This value includes the sizes of its parameter (4 bytes) and the minimum stack usage from the two branches (4 bytes).

Functions with Variable Number of Parameters (Variadic Functions)

```
#include <stdarg.h>

void fun_vararg(int x, ...) {
```

```
  va_list ap;
  va_start(ap, x);
  int i;
  for (i=0; i<x; i++) {
    int j = va_arg(ap, int);
  }
  va_end(ap);
}


void call_fun_vararg1(void) {
  long long int l = 0;
  fun_vararg(3, 4, 5, 6, l);
}


void call_fun_vararg2(void) {
  fun_vararg(1,0);
}
```

In this function, `fun_vararg` is a function with variable number of parameters. The minimum stack usage of `fun_vararg` takes into account the call to `fun_vararg` with the minimum number of arguments. The call with the minimum number of arguments is the call in `call_fun_vararg2` with two arguments (one for the fixed parameter and one for the variable parameter). The minimum stack usages are:

- `fun_vararg`: 20 bytes.

  This value takes into account:

  - The size of the fixed parameter `x` (4 bytes).
  - The sizes of the variable parameters from the call with the minimum number of parameters. In that call, there is only one variable argument of type `int` (4 bytes).
  - The sizes of the local variables `i`, `j` and `ap` (12 bytes). The size of the `va_list` variable uses the pointer size defined in the target (in this case, 4 bytes).
- `call_fun_vararg1`: 44 bytes.

  This value takes into account:

  - The stack size usage of `fun_vararg` with five arguments (36 bytes, of which 12 bytes are for the local variable sizes and 20 bytes are for the fixed and variable parameters of `fun_vararg`).
  - The size of local variable `l` (8 bytes).
- `call_fun_vararg2`: 20 bytes.

  Since `call_fun_vararg2` has no local variables, this value is the same as the stack size usage of `fun_vararg` with two arguments (20 bytes).

Metric Information

Group: Function
Acronym: `MIN_STACK`
HIS Metric: No

See Also

`Calculate code metrics (-code-metrics)`

Introduced in R2017b

Number of Call Levels

Maximum depth of nesting of control flow structures

Description

This metric specifies the maximum nesting depth of control flow statements such as `if`, `switch`, `for`, or `while` in a function. A function without control-flow statements has a call level 1.

The recommended upper limit for this metric is 4. For better readability of your code, try to enforce an upper limit for this metric.

To enforce limits on metrics, see “Compute Code Complexity Metrics”.

Examples

Function with Nested `if` Statements

```
int foo(int x,int y)
{
    int flag = 0;
    if (x <= 0)
        /* Call level 1*/
        flag = 1;
    else
    {
        if (x <= y )
            /* Call level 2*/
            flag = 1;
        else
            flag = -1;
    }
    return flag;
}
```

In this example, the number of call levels of `foo` is 2.

Function with Nesting of Different Control-Flow Statements

```
int foo(int x,int y, int bound)
{
    int count = 0;
    if (x <= y)
        /* Call level 1*/
        count = 1;
    else
        while(x>y) {
            /* Call level 2*/
            x--;
            if(count< bound) {
                /* Call level 3*/
                count++;
            }
        }
```

```
    return count;
}
```

In this example, the number of call levels of `foo` is 3.

Metric Information

Group: Function
Acronym: LEVEL
HIS Metric: Yes

See Also

Calculate code metrics (-code-metrics)

Number of Call Occurrences

Number of calls in function body

Description

This metric specifies the number of function calls in the body of a function.

Calls through a function pointer are not counted. Calls in unreachable code and calls to standard library functions are counted. `assert` is considered as a macro and not a function, so it is not counted.

Examples

Same Function Called Multiple Times

```
int func1(void);
int func2(void);

int foo() {
    return (func1() + func1()*func1() + 2*func2());
}
```

In this example, the number of call occurrences in `foo` is 4.

Function Called in a Loop

```
#include<stdio.h>

void fillArraySize10(int *arr) {
    for(int i=0; i<10; i++)
        arr[i]=getVal();
}

int getVal(void) {
    int val;
    printf("Enter a value:");
    scanf("%d", &val);
    return val;
}
```

In this example, the number of call occurrences in `fillArraySize10` is 1.

Recursive Function

```
#include <stdio.h>

void main() {
 int count;
 printf("How many numbers ?");
 scanf("%d",&count);
 fibonacci(count);
}
```

```
int fibonacci(int num)
{
   if ( num == 0 )
      return 0;
   else if ( num == 1 )
      return 1;
   else
      return ( fibonacci(num-1) + fibonacci(num-2) );
}
```

In this example, the number of call occurrences in `fibonacci` is 2.

Metric Information

Group: Function
Acronym: NCALLS
HIS Metric: No

See Also

Number of Called Functions | Calculate code metrics (-code-metrics)

Number of Called Functions

Number of callees of a function

Description

This metric specifies the number of callees of a function.

Calls through a function pointer are not counted. Calls in unreachable code and calls to standard library functions are counted. `assert` is considered as a macro and not a function, so it is not counted.

The recommended upper limit for this metric is 7. For more self-contained code, try to enforce an upper limit on this metric.

To enforce limits on metrics, see “Compute Code Complexity Metrics”.

Examples

Same Function Called Multiple Times

```
int func1(void);
int func2(void);

int foo() {
    return (func1() + func1()*func1() + 2*func2());
}
```

In this example, the number of called functions in `foo` is 2. The called functions are `func1` and `func2`.

Recursive Function

```
#include <stdio.h>

void main() {
 int count;
 printf("How many numbers ?");
 scanf("%d",&count);
 fibonacci(count);
}

int fibonacci(int num)
{
   if ( num == 0 )
      return 0;
   else if ( num == 1 )
      return 1;
   else
      return ( fibonacci(num-1) + fibonacci(num-2) );
}
```

In this example, the number of called functions in `fibonacci` is 1. The called function is `fibonacci` itself.

Metric Information

Group: Function
Acronym: CALLS
HIS Metric: Yes

See Also

Number of Call Occurrences | Number of Calling Functions | Calculate code metrics (-code-metrics)

Number of Calling Functions

Number of distinct callers of a function

Description

This metric measures the number of distinct callers of a function.

Calls through a function pointer are not counted. Calls in unreachable code are counted. Even if a caller calls a function more than once, it is counted only once when this metric is calculated.

The recommended upper limit for this metric is 5. For more self-contained code, try to enforce an upper limit on this metric.

To enforce limits on metrics, see “Compute Code Complexity Metrics”.

Examples

Same Function Calling a Function Multiple Times

```
#include <stdio.h>

int getVal() {
    int myVal;
    printf("Enter a value:");
    scanf("%d", &myVal);
    return myVal;
}

int func() {
    int val=getVal();
    if(val<0)
        return 0;
    else
        return val;
}

int func2() {
    int val=getVal();
    while(val<0)
        val=getVal();
    return val;
}
```

In this example, the number of calling functions for `getVal` is 2. The calling functions are `func` and `func2`.

Recursive Function

```
#include <stdio.h>

void main() {
 int count;
 printf("How many numbers ?");
```

```
 scanf("%d",&count);
 fibonacci(count);
}

int fibonacci(int num)
{
   if ( num == 0 )
      return 0;
   else if ( num == 1 )
      return 1;
   else
      return ( fibonacci(num-1) + fibonacci(num-2) );
}
```

In this example, the number of calling functions for `fibonacci` is 2. The calling functions are `main` and `fibonacci` itself.

Metric Information

Group: Function
Acronym: `CALLING`
HIS Metric: Yes

See Also

`Number of Called Functions` | `Calculate code metrics (-code-metrics)`

Number of Direct Recursions

Number of instances of a function calling itself directly

Description

This metric specifies the number of direct recursions in your project.

A direct recursion is a recursion where a function calls itself in its own body. If indirect recursions do not occur, the number of direct recursions is equal to the number of recursive functions.

The recommended upper limit for this metric is 0. To avoid the possibility of exceeding available stack space, do not use recursions in your code. To detect use of recursions, check for violations of `MISRA C:2012 Rule 17.2`.

Examples

Direct Recursion

```
int getVal(void);

void main() {
    int count = getVal(), total;
    assert(count > 0 && count <100);
    total = sum(count);
}

int sum(int val) {
    if(val<0)
        return 0;
    else
        return (val + sum(val-1));
}
```

In this example, the number of direct recursions is 1.

Metric Information

Group: Project
Acronym: `AP_CG_DIRECT_CYCLE`
HIS Metric: Yes

See Also

`MISRA C:2012 Rule 17.2` | `Calculate code metrics (-code-metrics)`

Number of Executable Lines

Number of executable lines in function body

Description

This metric measures the number of executable lines in a function body. When calculating the value of this metric, Polyspace excludes declarations without static initializers, comments, blank lines, braces or preprocessing directives.

If the function body contains a `#include` directive, the included file source code is also calculated as part of this metric.

This metric is not calculated for C++ templates.

Examples

Function with Declarations, Braces and Comments

```
void func(int);

int getSign(int arg) {
    int sign;
    if(arg<0) {
        sign=-1;
        func(-arg);
        /* func takes positive arguments */
    }
    else if(arg==0)
        sign=0;
    else {
        sign=1;
        func(arg);
    }
    return sign;
}
```

In this example, the number of executable lines of `getSign` is 9. The calculation excludes:

- The declaration `int sign;`.
- The comment `/* ... */`.
- The two lines with braces only.

Metric Information

Group: Function
Acronym: FXLN
HIS Metric: No

See Also

Number of Lines Within Body | Number of Instructions | Calculate code metrics (-code-metrics)

Number of Files

Number of source files

Description

This metric calculates the number of source files in your project.

Metric Information

Group: Project
Acronym: FILES
HIS Metric: No

See Also

`Number of Header Files` | `Calculate code metrics (-code-metrics)`

Number of Function Parameters

Number of function arguments

Description

This metric measures the number of function arguments.

If ellipsis is used to denote variable number of arguments, when calculating this metric, the ellipsis is not counted.

The recommended upper limit for this metric is 5. For less dependency between functions and fewer side effects, try to enforce an upper limit on this metric.

To enforce limits on metrics, see “Compute Code Complexity Metrics”.

Examples

Function with Fixed Arguments

```
int initializeArray(int* arr, int size) {
}
```

In this example, `initializeArray` has two parameters.

Function with Type Definition in Arguments

```
int getValueInLoc(struct {int* arr; int size;}myArray, int loc) {
}
```

In this example, `getValueInLoc` has two parameters.

Function with Variable Arguments

```
double average ( int num, ... )
{
    va_list arg;
    double sum = 0;

    va_start ( arg, num );

    for ( int x = 0; x < num; x++ )
    {
        sum += va_arg ( arg, double );
    }
    va_end ( arg);

    return sum / num;
}
```

In this example, `average` has one parameter. The ellipsis denoting variable number of arguments is not counted.

Metric Information

Group: Function
Acronym: PARAM
HIS Metric: Yes

See Also

`Calculate code metrics (-code-metrics)`

Number of Goto Statements

Number of `goto` statements

Description

This metric measures the number of `goto` statements in a function.

`break` and `continue` statements are not counted.

The recommended upper limit on this metric is 0. For better readability of your code, avoid `goto` statements in your code. To detect use of `goto` statements, check for violations of `MISRA C:2012 Rule 15.1`.

To enforce limits on metrics, see “Compute Code Complexity Metrics”.

Examples

Function with `goto` Statements

```
#define SIZE 10
int initialize(int **arr, int loc);
void printString(char *);
void printErrorMessage(void);
void printExecutionMessage(void);

int main()
{
    int *arrayOfStrings[SIZE],len[SIZE],i;
    for ( i = 0; i < SIZE; i++ )
    {
        len[i] = initialize(arrayOfStrings,i);
    }

    for ( i = 0; i < SIZE; i++ )
    {
        if(len[i] == 0)
            goto emptyString;
        else
            goto nonEmptyString;
        loop: printExecutionMessage();
    }

emptyString:
    printErrorMessage();
    goto loop;
nonEmptyString:
    printString(arrayOfStrings[i]);
    goto loop;
}
```

In this example, the function `main` has 4 `goto` statements.

Metric Information

Group: Function
Acronym: GOTO
HIS Metric: Yes

See Also

`Calculate code metrics (-code-metrics)`

Number of Header Files

Number of included header files

Description

This metric measures the number of header files in the project. Both directly and indirectly included header files are counted.

The metric gives a slightly higher number than the actual number of header files that you use because Polyspace® internal header files and header files included by those files are also counted. For the same reason, the metric can vary slightly even if you do not explicitly include new header files or remove inclusion of header files from your code. For instance, the number of Polyspace® internal header files can vary if you change your analysis options.

Metric Information

Group: Project
Acronym: `INCLUDES`
HIS Metric: No

See Also

`Number of Files` | `Calculate code metrics (-code-metrics)`

Number of Instructions

Number of instructions per function

Description

This metric measures the number of instructions in a function body.

The recommended upper limit for this metric is 50. For more modular code, try to enforce an upper limit for this metric.

To enforce limits on metrics, see “Compute Code Complexity Metrics”.

Computation Details

The metric is calculated using the following rules:

- A simple statement ending with a `;` is one instruction.

  If the statement is empty, it does not count as an instruction.
- A variable declaration counts as one instruction only if the variable is also initialized.
- Control flow statements such as `if`, `for`, `break`, `goto`, `return`, `switch`, `while`, `do-while` count as one instruction.
- The following do not count as instructions by themselves:
  - Beginning of a block of code

    For instance, the following counts as one instruction:

    ```
    {
        var = 1;
    }
    ```
  - Labels

    For instance, the following counts as two instructions. The `case` labels do not count as instructions.

    ```
    switch (1) {  // Instruction 1: switch
        case 0:
        case 1:
        case 2:
        default:
        break;    // Instruction 2: break
     }
    ```

Examples

Calculation of Number of Instructions

```
int func(int* arr, int size) {
    int i, countPos=0, countNeg=0, countZero = 0;
    for(i=0; i<size; i++) {
```

```
        if(arr[i] >0)
            countPos++;
        else if(arr[i] ==0)
            countZero++;
        else
            countNeg++;
    }
}
```

In this example, the number of instructions in `func` is 9. The instructions are:

1. `countPos=0`
2. `countNeg=0`
3. `countZero=0`
4. `for(i=0;i<size;i++) { ... }`
5. `if(arr[i] >=0)`
6. `countPos++`
7. `else if(arr[i]==0)`

   The ending `else` is counted as part of the `if-else` instruction.
8. `countZero++`
9. `countNeg++`

Note This metric is different from the number of executable lines. For instance:

- `for(i=0;i<size;i++)` has 1 instruction and 1 executable line.
- The following code has 1 instruction but 3 executable lines.

```
for(i=0;
    i<size;
    i++)
```

Metric Information

Group: Function
Acronym: STMT
HIS Metric: Yes

See Also

`Calculate code metrics (-code-metrics)`

Number of Lines

Total number of lines in a file

Description

This metric calculates the number of lines in a file. When calculating the value of this metric, Polyspace includes comments and blank lines.

This metric is calculated for source files and header files in the same folders as source files. If you want:

- The metric reported for other header files, change the default value of the option `Generate results for sources and (-generate-results-for)`.
- The metric not reported for header files at all, change the value of the option `Do not generate results for (-do-not-generate-results-for)` to `all-headers`.

Metric Information

Group: File
Acronym: `TOTAL_LINES`
HIS Metric: No

See Also

`Number of Lines Without Comment` | `Calculate code metrics (-code-metrics)`

Number of Lines Within Body

Number of lines in function body

Description

This metric calculates the number of lines in function body. When calculating the value of this metric, Polyspace includes declarations, comments, blank lines, braces and preprocessing directives.

If the function body contains a `#include` directive, the included file source code is also calculated as part of this metric.

This metric is not calculated for C++ templates.

Examples

Function with Declarations, Braces and Comments

```
void func(int);

int getSign(int arg) {
    int sign;
    if(arg<0) {
        sign=-1;
        func(-arg);
        /* func takes positive arguments */
    }
    else if(arg==0)
        sign=0;
    else {
        sign=1;
        func(arg);
    }
    return sign;
}
```

In this example, the number of executable lines of `getSign` is 13. The calculation includes:

- The declaration `int sign;`.
- The comment `/* ... */`.
- The two lines with braces only.

Metric Information

Group: Function
Acronym: `FLIN`
HIS Metric: No

See Also

`Number of Executable Lines` | `Calculate code metrics (-code-metrics)`

Number of Lines Without Comment

Number of lines of code excluding comments

Description

This metric calculates the number of lines in a file. When calculating the value of this metric, Polyspace excludes comments and blank lines.

This metric is calculated for source files and header files in the same folders as source files. If you want:

- The metric reported for other header files, change the default value of the option `Generate results for sources and (-generate-results-for)`.
- The metric not reported for header files at all, change the value of the option `Do not generate results for (-do-not-generate-results-for)` to `all-headers`.

Metric Information

Group: File
Acronym: `LINES_WITHOUT_CMT`
HIS Metric: No

See Also

`Number of Lines` | `Calculate code metrics (-code-metrics)`

Number of Local Non-Static Variables

Total number of local variables in function

Description

This metric provides the number of local variables in a function.

The metric excludes static variables. To find number of static variables, use the metric `Number of Local Static Variables`.

Examples

Non-Structured Variables

```
int flag();

int func(int param) {
  int var_1;
  int var_2;
  if (flag()) {
      int var_3;
      int var_4;
    } else {
      int var_5;
    }
}
```

In this example, the number of local non-static variables in `func` is 5. The number does not include the function arguments and return value.

Arrays and Structured Variables

```
typedef struct myStruct{
   char  arr1[50];
   char  arr2[50];
   int   val;
} myStruct;

void func(void) {
  myStruct var;
  char localArr[50];
}
```

In this example, the number of local non-static variables in `func` is 2: the structured variable `var` and the array `localArr`.

Variables in Class Methods

```
class Rectangle {
    int width, height;
  public:
    void set (int,int);
    int area (void);
```

```
} rect;

int Rectangle::area (void) {
    int temp;
    temp = width * height;
    return(temp);
}
```

In this example, the number of local non-static variables in `Rectangle::area` is 1: the variable `temp`.

Metric Information

Group: Function
Acronym: `LOCAL_VARS`
HIS Metric: No

See Also

`Number of Local Static Variables` | `Higher Estimate of Local Variable Size` | `Lower Estimate of Local Variable Size` | `Calculate code metrics (-code-metrics)`

Introduced in R2017a

Number of Local Static Variables

Total number of local static variables in function

Description

This metric provides the number of local static variables in a function.

Examples

Number of Static Variables

```
void func(void) {
  static int var_1 = 0;
  int var_2;
}
```

In this example, the number of static variables in `func` is 1. For examples of different types of variables, see `Number of Local Non-Static Variables`.

Metric Information

Group: Function
Acronym: `LOCAL_STATIC_VARS`
HIS Metric: No

See Also

`Higher Estimate of Local Variable Size` | `Number of Local Non-Static Variables` | `Calculate code metrics (-code-metrics)`

Introduced in R2017a

Number of Paths

Estimated static path count

Description

This metric measures the number of paths in a function.

The recommended upper limit for this metric is 80. If the number of paths is high, the code is difficult to read and can cause more orange checks. Try to limit the value of this metric.

To enforce limits on metrics, see “Compute Code Complexity Metrics”.

Computation Details

The number of paths is calculated according to these rules:

- If the statements in a function do not break the control flow, the number of paths is one.

  Even an empty statement such as `;` or empty block such as `{}` counts as one path.
- The number of paths for a control flow statement is calculated as follows:
  - `if-else if-else`: The number of paths is the sum of paths calculated in the `if` block, each `else if` block, and the concluding `else` block. When the concluding `else` block is omitted, the path count is increased by 1.

    For instance, the statement `if(..) {} else if(..) {} else {}` counts as three paths. The statement `if() {}` counts as two paths, one for the `if` block and one for the omitted `else` block.
  - `switch-case`: Every `case` with `break` statement adds one to the path count. The `default` statement counts as one path, even if it is omitted.

    For instance, the statement `switch (var) { case 1: .. break; case 2: .. break; default: .. }` counts as three paths.
  - `for`, `while`, and `do-while`: The number of paths is equal to the number of paths in the loop body + 1.

    For instance, the statement `while(0) {;}` counts as two paths.
  - Ternary operators: A statement with a ternary operator such as

    ```
    result = a > b ? a : b;
    ```

    is counted as one statement that does not break the control flow. The number of paths is considered as one.
- If more than one control flow statement are present in a sequence, the number of paths is the product of the path count for each control flow statement.

  For instance, if a function has three `for` loops and two `if-else` statements, the number of paths is $2 \times 2 \times 2 \times 2 \times 2 = 32$.

If many control flow statements are present in a function, the number of paths can be large. Nested control flow statements reduce the number of paths at the cost of increasing the depth of nesting. For an example, see “Function with Nested Control Flow Statements” on page 12-56.

- The software displays specific values in cases where the metric is not calculated:
  - If `goto` statements are present in the body of the function, Polyspace cannot calculate the number of paths. The software displays a metric value of -1.
  - If the number of paths reaches an internal limit, the calculation stops. The software displays this limit as the metric value. The limit is 9223372036854775807 (indicating the hexadecimal number 0x7fffffffffffffff).

Examples

Function with One Path

```
void func(int ch) {
    switch (ch)
    {
    case 1:
    case 2:
    case 3:
    case 4:
    default:
    }
}
```

In this example, `func` has one path.

Function with Control Flow Statement Causing Multiple Paths

```
void func(int ch) {
    switch (ch)
    {
    case 1:
        break;
    case 2:
        break;
    case 3:
        break;
    case 4:
        break;
    default:
    }
}
```

In this example, `func` has five paths. Apart from the path that goes through the `case`s and `default`, each `break` causes the creation of a new path.

Function with Nested Control Flow Statements

```
void func()
{
    int i = 0, j = 0, k = 0;
    for (i=0; i<10; i++)
    {
        for (j=0; j<10; j++)
```

```
        {
            for (k=0; k<10; k++)
            {
                if (i < 2 )
                    ;
                else
                {
                    if (i > 5)
                        ;
                    else
                        ;
                }
            }
        }
    }
}
```

In this example, `func` has six paths. The number is calculated as follows:

- The innermost `if-else` block counts as two paths.
- The outer `if-else` block counts as three paths, one path for the `if` block and the previous two paths for the `else` block.
- The innermost `for` loop counts as four paths, one path for the loop and the previous three paths for the `if-else` blocks.
- The next two outer loops add one path each.

Therefore, the number of paths in `func` is six.

Metric Information

Group: Function
Acronym: PATH
HIS Metric: Yes

See Also

`Calculate code metrics (-code-metrics)`

Number of Potentially Unprotected Shared Variables

Number of unprotected shared variables

Description

This metric measures the number of variables with the following properties:

- The variable is used in more than one task.
- At least one operation on the variable is not protected from interruption by operations in other tasks.

Examples

Unprotected Shared Variables

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        reset();
        inc();
        inc();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt();
    }
}

void main() {
}
```

In this example, `shared_var` is an unprotected shared variable if you specify `task` and `interrupt_handler` as entry points and do not specify protection mechanisms.

The operation `shared_var = INT_MAX` can interrupt the other operations on `shared_var` and cause unpredictable behavior.

Metric Information

Group: Project
Acronym: UNPSHV
HIS Metric: No

See Also

`Calculate code metrics (-code-metrics)`

Introduced in R2018b

Number of Protected Shared Variables

Number of protected shared variables

Description

This metric measures the number of variables with the following properties:

- The variable is used in more than one task.
- All operations on the variable are protected from interruption through critical sections or temporal exclusions.

Examples

Shared Variables Protected Through Temporal Exclusion

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        reset();
        inc();
        inc();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt();
    }
}

void main() {
}
```

In this example, `shared_var` is a protected shared variable if you specify the following options:

Option	Value

Entry points	`task`
	
	`interrupt_handler`

Temporally exclusive tasks	`task interrupt_handler`

The variable is shared between `task` and `interrupt_handler`. However, because `task` and `interrupt_handler` are temporally exclusive, operations on the variable cannot interrupt each other.

Shared Variables Protected Through Critical Sections

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void take_semaphore(void);
void give_semaphore(void);

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        take_semaphore();
        reset();
        inc();
        inc();
        give_semaphore();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        take_semaphore();
        interrupt();
        give_semaphore();
    }
}

void main() {
}
```

In this example, `shared_var` is a protected shared variable if you specify the following:

Option	Value	
Entry points	`task` `interrupt_handler`	
Critical section details	**Starting routine**	**Ending routine**
	`take_semaphore`	`give_semaphore`

The variable is shared between `task` and `interrupt_handler`. However, because operations on the variable are between calls to the starting and ending procedure of the same critical section, they cannot interrupt each other.

Metric Information

Group: Project
Acronym: PSHV
HIS Metric: No

See Also

`Calculate code metrics (-code-metrics)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Introduced in R2018b

Number of Recursions

Number of call graph cycles over one or more functions

Description

The metric provides a quantitative estimate of the number of recursion cycles in your project. The metric is the sum of:

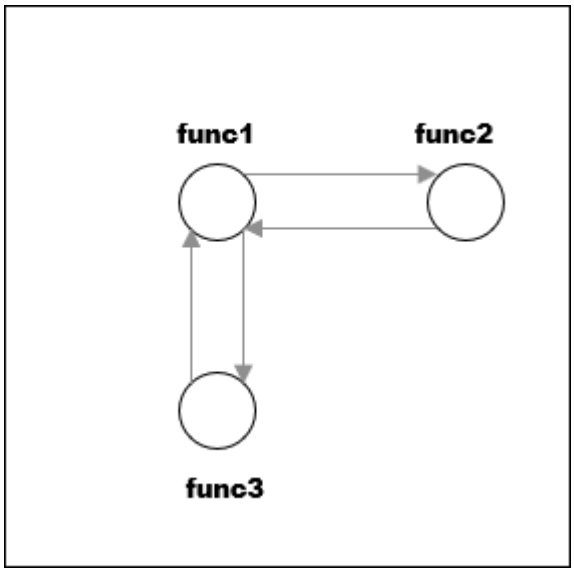
- Number of direct recursions (self recursive functions or functions calling themselves).
- Number of strongly connected components formed by the indirect recursion cycles in your project. If you consider the recursion cycles as a directed graph, the graph is strongly connected if there is a path between all pairs of vertices.

  To compute the number of strongly connected components:

  1 Draw the recursion cycles in your code.

  For instance, the recursion cycles in this example are shown below.

```
volatile int checkStatus;
void func1() {
   if(checkStatus) {
        func2();
   }
   else {
        func3();
   }
}

func2() {
   func1();
}

func3() {
   func1();
}
```



2 Identify the number of strongly connected components formed by the recursion cycles.

In the preceding example, there is one strongly connected component. You can move from any vertex to another vertex by following the paths in the graph.

The event list below the metric shows one of the recursion cycles in the strongly connected component.

Number of Recursions (Value: 1)
This metric shows the number of recursions, both direct and indirect.

	Event	File	Scope	Line
1	Recursion cycle: func1 => func3	file.c	file.c	2

Calls through a function pointer are not considered.

The recommended upper limit for this metric is 0. To avoid the possibility of exceeding available stack space, do not use recursions in your code. Recursions can tend to exhaust stack space easily. See examples of stack size growth with recursions described for this CERT-C rule that forbids recursions.

To detect use of recursions, check for violations of one of `MISRA C:2012 Rule 17.2`,MISRA C: 2004 Rule 16.2, `MISRA C++:2008 Rule 7-5-4` or JSF Rule 119. Note that:

- The rule checkers report each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported.
- The rule checkers consider explicit function calls only. For instance, in C++ code, the rule checkers ignore implicit calls to constructors during object creation. However, the metrics computation considers both implicit and explicit calls.

To enforce limits on metrics, see “Compute Code Complexity Metrics”.

Examples

Direct Recursion

```
int getVal(void);

void main() {
    int count = getVal(), total;
    assert(count > 0 && count <100);
    total = sum(count);
}

int sum(int val) {
    if(val<0)
        return 0;
    else
        return (val + sum(val-1));
}
```

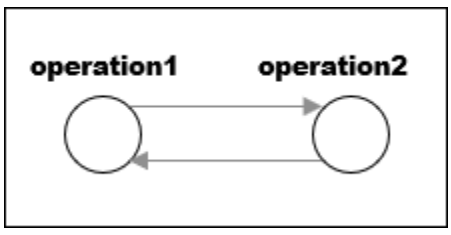
In this example, the number of recursions is 1.

A direct recursion is a recursion where a function calls itself in its own body. For direct recursions, the number of recursions is equal to the number of recursive functions.

Indirect Recursion with One Call Graph Cycle

```
volatile int signal;

void operation1() {
    int stop = signal%2;
    if(!stop)
        operation2();
}

void operation2() {
    operation1();
}

void main() {
    operation1();
}
```

In this example, the number of recursions is one. The two functions `operation1` and `operation2` are involved in the call graph cycle `operation1` → `operation2` → `operation1`.



An indirect function is a recursion where a function calls itself through other functions. For indirect recursions, the number of recursions can be different from the number of recursive functions.

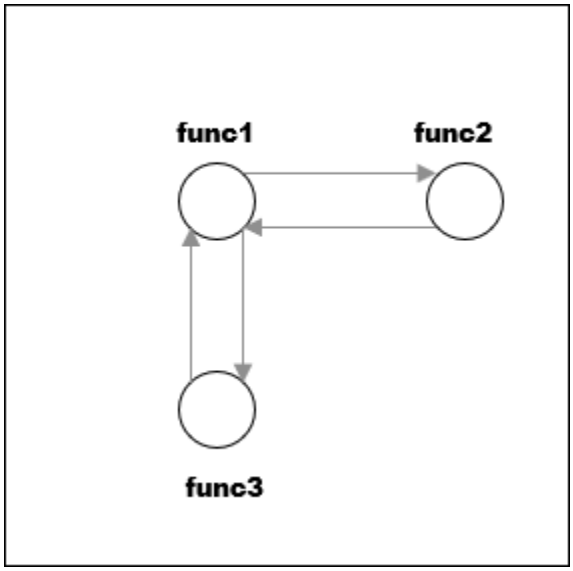
Multiple Call Graph Cycles Forming One Strongly Connected Component

```
volatile int checkStatus;
void func1() {
   if(checkStatus) {
        func2();
   }
   else {
        func3();
   }
}

func2() {
   func1();
}

func3() {
   func1();
}
```

In this example, there are two call graph cycles:

- `func1` → `func2` → `func1`
- `func1` → `func3` → `func1`

However, the cycles form one strongly connected component. You can move from any vertex to another vertex by following the paths in the graph. Hence, the number of recursions is one.



Indirect Recursion with Two Call Graph Cycles

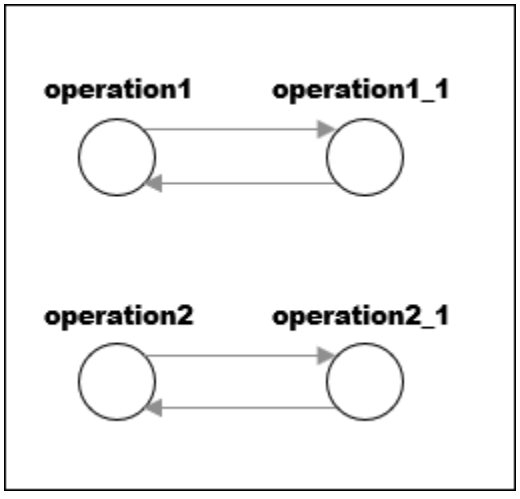
```
volatile int signal;

void operation1() {
    int stop = signal%2;
    if(!stop)
        operation1_1();
}


void operation1_1() {
    operation1();
}

void operation2() {
    int stop = signal%2;
    if(!stop)
        operation2_1();
}

void operation2_1() {
    operation2();
}

void main(){
    operation1();
    operation2();
}
```

In this example, the number of recursions is two.

There are two call graph cycles:

- `operation1` → `operation1_1` → `operation1`
- `operation2` → `operation2_1` → `operation2`

The call graph cycles form two strongly connected components.

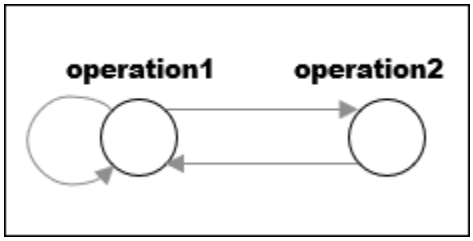


Same Function Called in Direct and Indirect Recursion

```
volatile int signal;

void operation1() {
    int stop = signal%3;
    if(stop==1)
        operation1();
    else if(stop==2)
        operation2();
}

void operation2() {
    operation1();
}

void main() {
    operation1();
}
```

In this example, the number of recursions is two:

- The strongly connected component formed by the cycle `operation1` → `operation2` → `operation1`.
- The self-recursive function `operation1`.



Metric Information

Group: Project
Acronym: `AP_CG_CYCLE`
HIS Metric: Yes

See Also

`MISRA C:2012 Rule 17.2` | `Calculate code metrics (-code-metrics)`

Number of Return Statements

Number of `return` statements in a function

Description

This metric measures the number of `return` statements in a function.

The recommended upper limit for this metric is 1. If one return statement is present, when reading the code, you can easily identify what the function returns.

To enforce limits on metrics, see “Compute Code Complexity Metrics”.

Examples

Function with Return Points

```
int getSign (int arg) {
    if(arg <0)
        return -1;
    else if(arg > 0)
        return 1;
    return 0;
}
```

In this example, `getSign` has 3 `return` statements.

Metric Information

Group: Function
Acronym: RETURN
HIS Metric: Yes

See Also

`Calculate code metrics (-code-metrics)`

Program Maximum Stack Usage

Maximum stack usage in the analyzed program

Description

This metric is reported in a Code Prover analysis only.

This metric shows the maximum stack usage from your program.

The metric shows the maximum stack usage for the function with the highest stack usage. If you provide a complete application, the function with the highest stack usage is typically the `main` function because the `main` function is at the top of the call hierarchy. For a description of maximum stack usage for a function, see the metric .

Metric Information

Group: Project
Acronym: `PROG_MAX_STACK`
HIS Metric: No

See Also

`Calculate code metrics (-code-metrics)`

Introduced in R2017b

Program Minimum Stack Usage

Maximum stack usage in the analyzed program taking nested scopes into account

Description

This metric is reported in a Code Prover analysis only.

This metric shows the maximum stack usage from your program, taking nested scopes into account.

The metric shows the minimum stack usage for the function with the highest stack usage. If you provide a complete application, the function with the highest stack usage is typically the `main` function because the `main` function is at the top of the call hierarchy. For a description of minimum stack usage for a function, see the metric .

Considering nested scopes is useful for compilers that reuse stack space for variables defined in nested scopes. For instance, in this code, the space for `var_1` is reused for `var_2`.

```
type func (type param_1, ...) {

  {
    /* Scope 1 */
    type var_1, ...;
  }
  {
    /* Scope 2 */
    type var_2, ...;
  }
}
```

Metric Information

Group: Project
Acronym: PROG_MIN_STACK
HIS Metric: No

See Also

`Calculate code metrics (-code-metrics)`

Introduced in R2017b

Report Components

Acronym Definitions

Create table of Polyspace acronyms used in report and their full forms

Description

This component creates a table containing the acronyms used in the report and their full forms. Acronyms are used for Polyspace checks and result status.

See Also

Topics
“Customize Existing Bug Finder Report Template”

Call Hierarchy

Create table showing call graph in source code

Description

This component creates a table showing the call hierarchy in your source code. For each function call in your source code, the table displays the following information:

- Level of call hierarchy, where the function is called.

  Each level is denoted by |. If a function call appears in the table as `|||->` *`file_name.function_name`*, the function call occurs at the third level of the hierarchy. Beginning from `main` or an entry point, there are three function calls leading to the current call.
- File containing the function call.

  In Code Prover, the line and column is also displayed.
- File containing the function definition.

  In Code Prover, the line and column where the function definition begins is also displayed.

In addition, the table also displays uncalled functions.

This table captures the information available on the **Call Hierarchy** pane in the Polyspace user interface.

See Also

Topics
“Customize Existing Bug Finder Report Template”

Code and Verification Information

Create table of verification times and code characteristics

Description

This component creates tables containing verification times and code characteristics such as number of lines.

Properties

Include Verification Time Information

If you select this option, the report contains verification times broken down by phase.

- For Polyspace Bug Finder, the phases are `compilation`, `pass0`, `pass1`, etc.
- For Polyspace Code Prover, the phases are `compilation`, `global`, `function`, etc.

Include Code Details

If you select this option, the report contains the following code characteristics:

- Number of files
- Number of lines
- Number of lines without comment

See Also

Topics
“Customize Existing Bug Finder Report Template”

Code Metrics Details

Create table of Polyspace metrics broken down by file and function

Description

This component creates a table containing metrics from a Polyspace project. The metrics appear broken down by file and function.

Properties

Project Metrics

If you select this option, the report contains the following metrics about the project:

- Number of direct recursions
- Number of files
- Number of headers
- Number of protected and unprotected shared variables

File Metrics

If you select this option, the report contains the following metrics about each file in the project:

- Estimated function coupling
- Lines without comment
- Comment density
- Total lines

Function Metrics

If you select this option, the report contains the following metrics about each function in the project:

- Cyclomatic complexity
- Language scope
- Lower and higher estimates of local variable size
- Number of lines within body
- Number of executable lines
- Number of `goto` statements
- Number of call levels
- Number of called functions
- Number of call occurrences
- Number of function parameters
- Number of paths
- Number of `return` statements

- Number of instructions
- Number of calling functions

See Also

Topics
“Customize Existing Bug Finder Report Template”

Code Metrics Summary

Create table of Polyspace metrics

Description

This component creates a table containing metrics from a Polyspace project. The metrics are the same as those displayed under `Code Metrics Details`. However, the file and function metrics are not broken down by individual files and functions. Instead, the table provides the minimum and maximum value of a file metric over all files and a function metric over all functions.

See Also

Topics
“Customize Existing Bug Finder Report Template”

Code Verification Summary

Create table of Polyspace analysis results

Description

This component creates tables containing the following results:

- Number of results
- Number of coding rule violations for each coding rule type such as MISRA C
- Number of defects, for Polyspace Bug Finder results
- Number of checks of each color, for Polyspace Code Prover results
- Whether the project passed or failed the software quality objective

Properties

Include Checks from Polyspace Standard Library Stub Functions

Unless you deselect this option, the tables contain Polyspace Code Prover checks that appear in Polyspace stubs for the standard library functions.

See Also

Topics
“Customize Existing Bug Finder Report Template”

Coding Rules Details

Create table of coding rule violations broken down by file

Description

This component creates tables containing coding rule violations broken down by each file in the Polyspace project. For each rule violation, the table contains the following information:

- Rule number
- Rule description
- Function containing the violation
- (Code Prover only) Line and column number
- Review information such as classification, status and comments

Properties

Select Coding Rules Type

Using this option, you can choose which coding rule violations to display. You can display violations for the following set of coding rules:

- MISRA C rules
- MISRA AC AGC rules
- MISRA C++ rules
- JSF C++ rules
- Custom coding rules

Display by

Using this option, you can break down the display of coding rule violations by file.

See Also

Topics
“Customize Existing Bug Finder Report Template”

Coding Rules Summary

Create table with number of coding rule violations

Description

This component creates a table containing the number of coding rule violations. You can choose whether to break this information down by rule number or file.

Properties

Select Coding Rules Type

Using this option, you can choose which coding rule violations to display. You can display violations for the following set of coding rules:

- MISRA C rules
- MISRA AC AGC rules
- MISRA C++ rules
- JSF C++ rules
- Custom coding rules

Include Files/Rules with No Problems Detected

If you select this option, the table displays:

- Files that do not contain coding rule violations
- Rules that your code does not violate

Display by

Using this option, you can break down the display of coding rule violations by:

- Rule number
- File

See Also

Topics

“Customize Existing Bug Finder Report Template”

Configuration Parameters

Create table of analysis options, assumptions and coding rules configuration

Description

This component creates the following tables:

- *Polyspace settings*: The analysis options that you used to obtain your results. The table lists command-line version of the options along with their values.
- *Analysis assumptions*: The assumptions used to obtain your Code Prover results. The table lists only the modifiable assumptions. For assumptions that you cannot change, see the Polyspace documentation.
- *Coding rules configuration*: The coding rules whose violations you checked for. The table lists the rule number, rule description and other information about the rules.
- *Files with compilation errors*: If your project has source files with compilation errors, these files are listed.

See Also

Topics
“Customize Existing Bug Finder Report Template”

Defects Summary

Create table of defects (Bug Finder only)

Description

This component creates a table of Polyspace Bug Finder defects. From this table, you can see the number of defects of each type.

Properties

Include Checkers with No Defects Detected

If you select this option, the table includes all defect types that Polyspace Bug Finder can detect, including those that do not occur in your code.

See Also

Topics
“Customize Existing Bug Finder Report Template”

Global Variable Checks

Create table of global variables (Code Prover only)

Description

This component creates a table of Polyspace Code Prover global variables. From this table, you can see the number of global variables of each type.

See Also

Topics
“Customize Existing Bug Finder Report Template”

Recursive Functions

Create table of recursive functions

Description

This component creates a table containing the recursive functions in your source code (along with the files containing the functions).

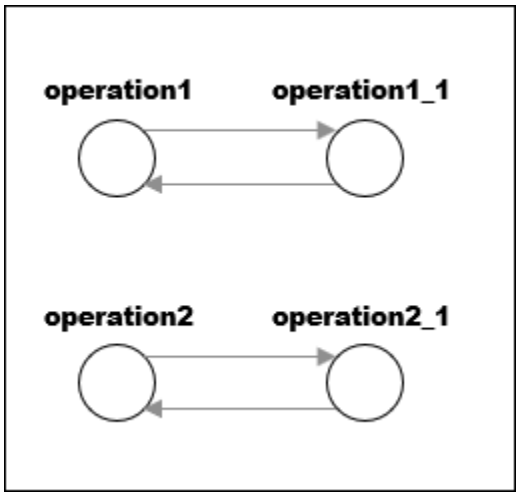
- For each direct recursion (function calling itself directly), the table lists the recursive function.
- For each indirect recursion cycle (function calling itself through other functions), the table lists one function in the cycle.

For instance, the following code contains two indirect recursion cycles.

```
volatile int signal;

void operation1() {
    int stop = signal%2;
    if(!stop)
        operation1_1();
}


void operation1_1() {
    operation1();
}

void operation2() {
    int stop = signal%2;
    if(!stop)
        operation2_1();
}

void operation2_1() {
    operation2();
}

void main(){
    operation1();
    operation2();
}
```

The two call graph cycles are:

- `operation1` → `operation1_1` → `operation1`
- `operation2` → `operation2_1` → `operation2`



This report component shows one function from each of the two cycles: `operation1` and `operation2`. To see the full cycle, open the results in the Polyspace user interface.

See Also

Topics
“Customize Existing Bug Finder Report Template”

Report Customization (Filtering)

Create filters that apply to your Polyspace reports

Description

This component allows you to filter unwanted information from existing Polyspace report templates. To apply global filters, place this component immediately below the node representing the report name.

Properties

Code Metrics Filters

The properties in table below apply to the inclusion of code metrics in your report.

Property	Purpose	User Action
Include Project Metrics	Choose whether to include metrics about your Polyspace project.	Select the check box to include project metrics.
Project metrics to include	Specify project metrics to include or exclude from report.	Enter a MATLAB regular expression.
Include File Metrics	Choose whether to include per file metrics in report.	Select the check box to include per file metrics.
File Metrics > Files to include	Specify files to include or exclude when reporting file metrics.	Enter a MATLAB regular expression.
File metrics to include	Specify file metrics to include or exclude from report.	Enter a MATLAB regular expression.
Include Function Metrics	Choose whether to include per function metrics in report.	Select the check box to include per function metrics.
Function Metrics > Files to include	Specify files to include or exclude when reporting function metrics.	Enter a MATLAB regular expression.
Functions to include	Specify functions to include or exclude when reporting function metrics.	Enter a MATLAB regular expression.
Function metrics to include	Specify function metrics to include or exclude from report.	Enter a MATLAB regular expression.

Coding Rules Filters

The properties in table below apply to the inclusion of coding rule violations in your report.

Property	Purpose	User Action
Files to include	Specify files to include or exclude when reporting coding rule violations.	Enter a MATLAB regular expression.
Coding rule numbers to include	Specify coding rules to include or exclude when reporting coding rule violations.	Enter a MATLAB regular expression.
Classifications to include	Specify classifications to include or exclude when reporting coding rule violations.	Enter a MATLAB regular expression.
Status types to include	Specify statuses to include or exclude when reporting coding rule violations.	Enter a MATLAB regular expression.

Run-time Check Filters

The properties in table below apply to the inclusion of Polyspace Code Prover checks in your report.

Property	Purpose
Red Checks	Specify whether to include red checks in your report. Red checks indicate proven run-time errors.
Gray Checks	Specify whether to include gray checks in your report. Gray checks indicate unreachable code.
Orange Checks	Specify whether to include orange checks in your report. Orange checks indicate possible run-time errors.
Green Checks	Specify whether to include green checks in your report. Green checks indicate that an operation does not contain a specific run-time error.
Inspection Point Checks	Specify whether to include inspection point checks in your report. These checks allow an user to find the values that a variable can take at a certain point in the code.
Unreachable Functions	Specify whether to include unreachable functions in your report.

Advanced Filters

The properties in table below apply to the inclusion of metrics, coding rule violations and Polyspace Code Prover checks in your report.

Property	Purpose	User Action
Justification status	Choose whether to report only justified checks, only unjustified checks or all checks.	Choose an option from the dropdown list.

Property	Purpose	User Action
Files to include	Specify files to include or exclude from your report.	Enter a MATLAB regular expression.
Check types to include	Specify Polyspace Code Prover checks to include in your report.	Enter a MATLAB regular expression.
Function names to include	Specify functions to include or exclude from your report.	Enter a MATLAB regular expression.
Classification types to include	Specify classifications to include or exclude from your report.	Enter a MATLAB regular expression.
Status types to include	Specify statuses to include or exclude from your report.	Enter a MATLAB regular expression.
Comments to include	Specify comments to include or exclude from your report.	Enter a MATLAB regular expression.

See Also

Topics
“Customize Existing Bug Finder Report Template”
“Regular Expressions” (MATLAB)

Run-time Checks Details Ordered by Color/File

Create overrides for global filters in Polyspace reports (Code Prover only)

Description

This component adds detailed information about the run-time checks to your report. This component can also be used to override global filters in specific chapters of your report. Use the following workflow when using filters in your report:

1. To create filters that apply to all chapters of your report, use the **Report Customization (Filtering)** component. For more information, see `Report Customization (Filtering)`.
2. To override some of the filters in individual chapters, use the **Run-time Checks Details Ordered by Color/File** component. Select the **Override Global Report filter** box.

Properties

Categories To Include

The properties in table below apply to the inclusion of Polyspace Code Prover checks in your report.

Property	Purpose
Red Checks	Specify whether to include red checks in your report. Red checks indicate proven run-time errors.
Gray Checks	Specify whether to include gray checks in your report. Gray checks indicate unreachable code.
Orange Checks	Specify whether to include orange checks in your report. Orange checks indicate possible run-time errors.
Green Checks	Specify whether to include green checks in your report. Green checks indicate that an operation does not contain a specific run-time error.
Inspection Point Checks	Specify whether to include inspection point checks in your report. These checks allow an user to find the values that a variable can take at a certain point in the code.
Unreachable Functions	Specify whether to include unreachable functions in your report.

Advanced Filters

The properties in table below apply to the inclusion of metrics, coding rule violations and Polyspace Code Prover checks in your report.

Property	Purpose	User Action
Justification status	Choose whether to report only justified checks, only unjustified checks or all checks.	Choose an option from the dropdown list.
Files to include	Specify files to include or exclude from your report.	Enter a regular MATLAB expression.
Check types to include	Specify Polyspace Code Prover checks to include in your report.	Enter a regular MATLAB expression.
Function names to include	Specify functions to include or exclude from your report.	Enter a regular MATLAB expression.
Classification types to include	Specify classifications to include or exclude from your report.	Enter a regular MATLAB expression.
Status types to include	Specify statuses to include or exclude from your report.	Enter a regular MATLAB expression.
Comments to include	Specify comments to include or exclude from your report.	Enter a regular MATLAB expression.

See Also

Topics

“Customize Existing Bug Finder Report Template”

Run-time Checks Details Ordered by Review Information

Create table with run-time checks ordered by review information (Code Prover only)

Description

This component creates tables displaying the Polyspace Code Prover checks in your code. All checks with same combination of **Severity** and **Status** appear in the same table.

See Also

Topics
“Customize Existing Bug Finder Report Template”

Run-time Checks Summary Ordered by File

Create table with run-time checks ordered by file (Code Prover only)

Description

This component creates a table displaying the number of Polyspace Code Prover checks per file in your code.

Properties

Sort the data

Use this option to sort the rows in the table alphabetically by filename or by percentage of unproven code.

Display as

Use this option to display the number of checks in a table or in bar charts.

Display ratio of checks in a file

Select this option to display the number of checks of a certain color as a ratio of total number of checks in the file.

Include checks from Polyspace standard library stub functions

Select this option to include the checks from Polyspace standard library stub functions in your display.

See Also

Topics
“Customize Existing Bug Finder Report Template”

Software Quality Objectives - Coding Rules Summary

Create table of coding rule violations in results downloaded from Polyspace Metrics

Description

This component creates a table containing coding rule violations in results downloaded from Polyspace Metrics.

See Also

Topics
“Customize Existing Bug Finder Report Template”

Software Quality Objectives - Run-time Checks Details

Create table of result details for results downloaded from Polyspace Metrics

Description

This component creates tables showing results downloaded from Polyspace Metrics.

The component `Software Quality Objectives - Run-time Checks Summary` shows the distribution of results. This component shows individual instances of results. Each file has a dedicated table showing the findings in the file.

See Also

Topics
“Customize Existing Bug Finder Report Template”

Software Quality Objectives - Run-time Checks Summary

Create table of results summary for results downloaded from Polyspace Metrics

Description

This component creates a table showing the distribution of run-time checks in results downloaded from Polyspace Metrics.

This component shows the distribution of run-time checks. The component `Software Quality Objectives - Run-time Checks Details` shows the individual instances of run-time checks.

See Also

Topics
“Customize Existing Bug Finder Report Template”

Summary By File

Create table showing summary of Polyspace results by file

Description

This component creates a table showing a breakdown of Polyspace results by file.

See Also

Topics
“Customize Existing Bug Finder Report Template”

Variable Access

Create table showing global variable access in source code (Code Prover only)

Description

This component creates a table showing the global variable access in your source code. For each global variable, the table displays the following information:

- Variable name.

  The entry for each variable is denoted by |.
- Type of the variable.
- Number of read and write operations on the variable.
- Details of read and write operations. For each read or write operation, the table displays the following information:
  - File and function containing the operation in the form *`file_name.function_name`*.

    The entry for each read or write operation is denoted by `||`. Write operations are denoted by < and read operations by >.
  - Line and column number of the operation.

This table captures the information available on the **Variable Access** pane in the Polyspace user interface.

The table showing variable access contains only the names of files. Below this table, a second table shows the full paths to files (in two columns, **Filename** and **Full filename**). If a variable access occurs in a Standard library function, the two columns contain this information:

- **Filename**: `__polyspace__stdstubs.c` (the file containing Polyspace implementation of Standard Library functions)
- **Full filename**: `Std library`

See Also

Topics
“Customize Existing Bug Finder Report Template”

Variable Checks Details Ordered By Review Information

Create table with global variable results ordered by review information (Code Prover only)

Description

This component creates tables displaying the Polyspace Code Prover global variable results in your code. All checks with same combination of **Severity** and **Status** appear in the same table.

See Also

Topics
“Customize Existing Bug Finder Report Template”

Configuration Parameters

Settings from (C)

Select settings for the analysis configuration. You can quickly activate coding rules checking for generated C code

Model Configuration Parameters Category: Polyspace

Settings

Default: `Project configuration`

`Project configuration`

Run Polyspace with the options specified in the “Project configuration” on page 14-7 or “Use custom project file” on page 14-6.

You do not check coding rules unless you select a rule set in the configuration.

`Project configuration and MISRA AC AGC checking`

Run Polyspace with the options specified in the **Project configuration** plus MISRA AC-AGC obligatory and recommended rules.

`Project configuration and MISRA C 2004 checking`

Run Polyspace with the options specified in the **Project configuration** plus all MISRA C 2004 rules.

`Project configuration and MISRA C 2012 checking`

Run Polyspace with the options specified in the **Project configuration** plus all MISRA C 2012 rules. This option automatically applies the rule categories for generated code. See `Use generated code requirements (-misra3-agc-mode)`.

`MISRA AC AGC checking`

Check compliance with the MISRA AC-AGC obligatory and recommended rules. After rules checking, Polyspace stops.

`MISRA C 2004 checking`

Check compliance with all MISRA C 2004 rules. After rules checking, Polyspace stops.

`MISRA C 2012 checking`

Check compliance with all MISRA C 2012 rules. This option automatically applies the rule categories for generated code. See `Use generated code requirements (-misra3-agc-mode)`. After rules checking, Polyspace stops.

Dependency

This setting overrides custom configuration settings in “Project configuration” on page 14-7 and “Use custom project file” on page 14-6. If you want to use your custom coding rule settings, select the `Project configuration` option.

Command-Line Information

Use the `pslinkoptions` property `VerificationSettings`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model.

Use the parameter `PSVerificationSettings` with the same value as for the `pslinkoptions` property `VerificationSettings`. See pslinkoptions.

See Also

pslinkoptions | `pslinkoptions`

Related Examples

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

Settings from (C++)

Select settings for the analysis configuration. This option allows you to quickly activate coding rules checking for generated C++ code.

Model Configuration Parameters Category: Polyspace

Settings

Default: `Project configuration`

`Project configuration`

Run Polyspace with the options specified in the “Project configuration” on page 14-7 or “Use custom project file” on page 14-6.

You do not check coding rules unless you select a rule set in the configuration.

`Project configuration and MISRA C++ checking`

Run Polyspace with the options specified in the **Project configuration** plus MISRA C++ required rules.

`Project configuration and JSF C++ checking`

Run Polyspace with the options specified in the **Project configuration** plus JSF C++ shall rules.

`MISRA C++ checking`

Check compliance with the MISRA C++: 2008 required rules. After rules checking, Polyspace stops.

`JSF C++ checking`

Check compliance with the JSF C++ shall rules. After rules checking, Polyspace stops.

Dependency

This setting overrides custom configuration settings in “Project configuration” on page 14-7 and “Use custom project file” on page 14-6. If you want to use your custom coding rule settings, select the `Project configuration` option.

Command-Line Information

Use the `pslinkoptions` property `CxxVerificationSettings`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSCxxVerificationSettings` with the same value as for the `pslinkoptions` property `CxxVerificationSettings`. See pslinkoptions.

See Also

`pslinkoptions` | pslinkoptions

Related Examples

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

Use custom project file

Set Polyspace configuration options with a custom `.psprj` file

Model Configuration Parameters Category: Polyspace

Settings

Default: Off ☐

Off ☐

Analysis uses configuration options from **Project configuration** on page 14-7 parameters.

On ☑

Analysis uses configuration options from the specified `.psprj` project file.

Dependency

The **Settings from** parameter overrides custom configuration settings for coding rules. If you want to use your custom coding rule settings, set **Settings from** > **Project configuration**.

Command-Line Information

Use the `pslinkoptions` properties `EnablePrjConfigFile` and `PrjConfigFile`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameters `PSEnablePrjConfigFile` and `PSPrjConfigFile` with the same values as for the `pslinkoptions` properties `EnablePrjConfigFile` and `PrjConfigFile`. See pslinkoptions.

See Also

`pslinkoptions` | pslinkoptions

Related Examples

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

Project configuration

Set advanced configuration options to customize the analysis.

Settings

Open the Polyspace Configuration window by using the **Configure** button. Customize additional settings in this window and save your project configuration. If you added a custom project file in the parameter “Use custom project file” on page 14-6, that project file configuration is shown. Otherwise, the default project template is used.

For details about the advanced options, see “Analysis Options”.

Dependency

The **Settings from** parameter overrides custom configuration settings for coding rules. If you want to use your custom coding rule settings, set **Settings from** > **Project configuration**.

Command-Line Information

Use a Polyspaceproject (`.psprj` file) with the `pslinkoptions` properties `EnablePrjConfigFile` and `PrjConfigFile`.

See Also

`polyspace.ModelLinkOptions` | `pslinkoptions` | pslinkoptions

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

Enable additional file list

Add additional supporting code files to the analysis.

For instance, suppose you use C files for testing results from the generated code or providing inputs to the generated code. The analysis of generated code only considers files generated from the Simulink model. If you want the analysis to consider the C files that you use for testing or inputs, provide them as additional files.

Model Configuration Parameters Category: Polyspace

Settings

Default: Off ☐

Off ☐

The analysis includes no additional files.

On ☑

Polyspace analyzes the specified C/C++ files with the generated code. Use the **Select files** button to specify these additional files.

Command-Line Information

Use the `pslinkoptions` properties `EnableAdditionalFileList` and `AdditionalFileList`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameters `PSEnableAdditionalFileList` and `PSAdditionalFileList` with the same values as for the `pslinkoptions` properties `EnableAdditionalFileList` and `AdditionalFileList`. See pslinkoptions.

See Also

`pslinkoptions` | pslinkoptions

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

Stub lookup tables

Specify that the verification must stub auto-generated functions that use certain kinds of lookup tables in their body. The lookup tables in these functions use linear interpolation and do not allow extrapolation. That is, the result of using the lookup table always lies between the lower and upper bounds of the table.

If you use this option, the verification is more precise and has fewer orange checks. The verification of lookup table functions is usually imprecise. The software has to make certain assumptions about these functions. To avoid missing a run-time error, the verification assumes that the result of using the lookup table is within the full range allowed by the result data type. This assumption can cause many unproven results (orange checks) when a lookup table function is called. By using this option, you narrow down the assumption. For functions using lookup tables with linear interpolation and no extrapolation, the result is at least within the bounds of the table.

The option is relevant only if your model uses Lookup Table blocks.

Model Configuration Parameters Category: Polyspace

Settings

Default: On ☑

On ☑

For autogenerated functions that use lookup tables with linear interpolation and no extrapolation, the verification:

- Does not check for run-time errors in the function body.
- Calls a function stub instead of the actual function at the function call sites. The stub ensures that the result of using the lookup table is within the bounds of the table.

To identify if the lookup table in the function uses linear interpolation and no extrapolation, the verification uses information provided by the code generation product. For instance, if you use Embedded Coder to generate code, the lookup table functions with linear interpolation and no extrapolation follow specific naming conventions.

Off ☐

The verification does not stub autogenerated functions that use lookup tables.

Tips

- The option applies only to autogenerated functions. If you integrate your own C/C++ S-Function using lookup tables with the model, the option does not cause them to be stubbed.
- The option is on by default. For certification purposes, if you want your verification tool to be independent of the code generation tool, turn off the option.

Command-Line Information

Use the `pslinkoptions` property `AutoStubLUT`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model.

Use the parameter `PSAutoStubLUT` with the same value as for the `pslinkoptions` property `AutoStubLUT`. See pslinkoptions.

See Also

`pslinkoptions` | pslinkoptions

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

Input

Choose whether to constrain Inport block variables.

Model Configuration Parameters Category: Polyspace

Settings

Default: `Use specified minimum and maximum values`

`Use specified minimum and maximum values`

Analysis assumes minimum and maximum values for input variables. These values are specified in the Inport block dialog box. Use this value to reduce the number of false positive results.

`Unbounded inputs`

Analysis assumes full range for input variables. Use this value to run a robust analysis that includes values outside the expected range.

Command-Line Information

Use the `pslinkoptions` property `InputRangeMode`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSInputRangeMode` with the same value as for the `pslinkoptions` property `InputRangeMode`. See pslinkoptions.

See Also

`pslinkoptions` | pslinkoptions

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”
- “External Constraints on Polyspace Analysis of Generated Code”

Tunable parameters

Choose how to treat tunable parameter values during the analysis. Treat values as either constants or a range of values.

Model Configuration Parameters Category: Polyspace

Settings

Default: `Use calibration data`

`Use calibration data`

Analysis assumes constant values for tunable parameters. Use this value to run a contextual analysis. This option can reduce the number of false positive results.

`Use specified minimum and maximum values`

Analysis assumes a range of values for the tunable parameter variables. Specify maximum and minimum values in the model. Use this option to run a robust analysis that includes values outside the expected parameter value.

Command-Line Information

Use the `pslinkoptions` property `ParamRangeMode`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSParamRangeMode` with the same value as for the `pslinkoptions` property `ParamRangeMode`. See pslinkoptions.

See Also

`pslinkoptions` | pslinkoptions

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”
- “External Constraints on Polyspace Analysis of Generated Code”

Output

Choose whether to verify output values.

Code Prover option only. Bug Finder cannot check output values.

Model Configuration Parameters Category: Polyspace

Settings

Default: `No verification`

`No verification`

Polyspace does not verify output values.

`Verify outputs are within minimum and maximum values`

Polyspace checks to see if the output variable values are within the expected minimum and maximum values. Specify the minimum and maximum values in the output block dialog boxes.

Command-Line Information

Use the `pslinkoptions` property `OutputRangeMode`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSOutputRangeMode` with the same value as for the `pslinkoptions` property `OutputRangeMode`. See pslinkoptions.

See Also

`pslinkoptions` | pslinkoptions

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”
- “External Constraints on Polyspace Analysis of Generated Code”

Model reference verification depth

Only for models that use Embedded Coder generated code. Indicate how deep into the model hierarchy to analyze.

Model Configuration Parameters Category: Polyspace

Settings

Default: `Current model only`

`Current model only`

Polyspace analyzes only the current model

`1`

Polyspace analyzes the current model and the referenced models that are one level below the current model.

`2`

Polyspace analyzes the current model and the referenced models that are up to two levels below the current model.

`3`

Polyspace analyzes the current model and the referenced models that are up to three levels below the current model.

`All`

Polyspace analyzes the current model and all referenced models.

Command-Line Information

Use the `pslinkoptions` property `ModelRefVerifDepth`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSModelRefVerifDepth` with the same value as for the `pslinkoptions` property `ModelRefVerifDepth`. See pslinkoptions.

See Also

`pslinkoptions` | pslinkoptions

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

Model by model verification

Only for models that use Embedded Coder generated code. Analyze each model or referenced model individually. If you have a large project, this option can help modularize your analysis .

Model Configuration Parameters Category: Polyspace

Settings

Default: Off ☐

Off ☐

Polyspace analyzes your models together. Model interactions are analyzed.

On ☑

Polyspace analyzes your model and each of its referenced models in isolation. This option does not analyze model interactions.

Command-Line Information

Use the `pslinkoptions` property `ModelRefByModelRefVerif`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSModelRefByModelRefVerif` with the same value as for the `pslinkoptions` property `ModelRefByModelRefVerif`. See pslinkoptions.

See Also

`pslinkoptions` | pslinkoptions

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

Output folder

Specify the location and folder name for your analysis results.

Model Configuration Parameters Category: Polyspace

Settings

Default: `results_$ModelName$`

Enter a path for your results folder. If you do not use a full path, the results folder is relative to your current MATLAB folder.

If you select “Add results to current Simulink project” on page 14-18, the results folder is relative to the Simulink project folder.

By default, the software stores your results in *`Current Folder`*`\results_`*`model_name`*.

Command-Line Information

Use the `pslinkoptions` property `ResultDir`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSResultDir` with the same value as for the `pslinkoptions` property `ResultDir`. See pslinkoptions.

See Also

`pslinkoptions` | pslinkoptions

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

Make output folder name unique by adding a suffix

Add a unique suffix to the results folder for every run to avoid overwriting previous results.

Model Configuration Parameters Category: Polyspace

Settings

Default: Off ☐

Off ☐

Every time you rerun your analysis, your results are overwritten.

On ☑

For each run of the analysis, Polyspace specifies a new location for the results folder by appending a unique number to the folder name.

Command-Line Information

Use the `pslinkoptions` property `AddSuffixToResultDir`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSAddSuffixToResultDir` with the same value as for the `pslinkoptions` property `AddSuffixToResultDir`. See pslinkoptions.

See Also

`pslinkoptions` | pslinkoptions

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

Add results to current Simulink project

Add your Polyspace results to the current Simulink project. To use this option, you must have a Simulink project open.

Model Configuration Parameters Category: Polyspace

Settings

Default: Off ☐

Off ☐

Results are saved to the current folder.

On ☑

Results are saved to the currently open Simulink project.

Dependencies

You must have a Simulink project open to use this option.

Command-Line Information

Use the `pslinkoptions` property `AddToSimulinkProject`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSAddToSimulinkProject` with the same value as for the `pslinkoptions` property `AddToSimulinkProject`. See pslinkoptions.

See Also

`pslinkoptions` | pslinkoptions

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

Open results automatically after verification

Decide whether to open your results in the Polyspace interface after running analysis from Simulink.

Model Configuration Parameters Category: Polyspace

Settings

Default: On ☑

On ☑

After you run an analysis, your results open automatically in the Polyspace interface.

Off ☐

You must manually open your results after running an analysis.

Command-Line Information

Use the `pslinkoptions` property `OpenProjectManager`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSOpenProjectManager` with the same value as for the `pslinkoptions` property `OpenProjectManager`. See pslinkoptions.

See Also

`pslinkoptions` | pslinkoptions

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

Check configuration before verification

Check whether model and code configurations are optimal for code analysis.

Model Configuration Parameters Category: Polyspace

Settings

Default: `On (proceed with warnings)`

`On (proceed with warnings)`

The process stops for errors, but continues the code analysis if the configuration has only warnings.

`On (stop for warnings)`

If the configuration has errors or warnings, the process stops.

`Off`

The software does not check the configuration.

Command-Line Information

Use the `pslinkoptions` property `CheckConfigBeforeAnalysis`. For details, see pslinkoptions.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSVerifALLSFcnInstances` with the same value as for the `pslinkoptions` property `VerifALLSFcnInstances`. See pslinkoptions.

See Also

`pslinkoptions`

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

Verify all S-function occurrences

For S-Function analyses only. Run an analysis on all instances of the selected S-Function.

Model Configuration Parameters Category: Polyspace

Settings

Default: Off ☐

Off ☐

Analyze only the selected S-Function block. The analysis includes only information from the selected S-Function block.

On ☑

Analyze all occurrences of the S-function in the model. If the S-Function is included in the model multiple times, information from all occurrences is included in the analysis.

Command-Line Information

Use the `pslinkoptions` property `VerifALLSFcnInstances`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSVerifALLSFcnInstances` with the same value as for the `pslinkoptions` property `VerifALLSFcnInstances`. See pslinkoptions.

See Also

`pslinkoptions` | pslinkoptions

More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

15

Approximations Used During Bug Finder Analysis

Inputs in Polyspace Bug Finder

A Bug Finder analysis by default does not return a defect caused by a special value of an unknown input, unless the input is bounded. Polyspace makes no assumption about the value of unbounded inputs when your source code is incomplete. For example, in the following code Bug Finder detects a **division by zero** in `foo_1()`, but not in `foo_2()`:

```
int foo_1(int p)
{
  int x = 0;
  if ( p > -10 && p < 10 )  /* p is bounded by if statement */
      x = 100/p;    /* Division by zero detected */

  return x;
}

int foo_2(int p)    /* p is unbounded */
{
  int x = 0;
  x = 100/p;    /* Division by zero not detected */


  return x;
}
```

To set bounds on your input, add constraints in your code such as `assert` or `if`. At the cost of a possibly longer runtime, you can perform a more exhaustive analysis where all values of function inputs are considered when showing defects. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

See Also

“Global Variables in Polyspace Bug Finder” on page 15-3 | “Bug Finder Analysis Assumptions”

Global Variables in Polyspace Bug Finder

When you run a Bug Finder analysis, Polyspace makes certain assumptions about the initialization of global variables. These assumptions depend on how you declare and define global variables. For example, in this code

```
int foo(void) {
    return 1/gvar;
}
```

Bug Finder detects a **division by zero** defect with the variable `gvar` in these cases:

- You define `int gvar;` in the source code and provide a `main` function that calls `foo`. Bug Finder follows ANSI standards that state the variable is initialized to zero.
- You define `int gvar;` or declare `extern int gvar;` in the source code. Another function calls `foo` and sets `gvar=0`. Otherwise, when your source files are incomplete and do not contain a `main` function, Bug Finder makes no assumption about the initialization of `gvar`.
- You declare `const int gvar;`. Bug Finder assumes `gvar` is initialized to zero due to the `const` keyword.

See Also

“Inputs in Polyspace Bug Finder” on page 15-2 | “Bug Finder Analysis Assumptions”