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.
3 Apple Hill Drive
Natick, MA 01760-2098

Polyspace® Bug Finder™ Reference

© COPYRIGHT 2013–2018 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

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)

Contents

1 Option Descriptions

2 Polyspace Command-Line Options

3 Defects

4 Functions, Properties, Classes, and Apps

5 MISRA C 2012

6 Custom Coding Rules

Group 1: Files 6-2

Group 2: Preprocessing 6-3

Group 3: Type de nitions 6-4

Group 4: Structures 6-5

Group 5: Classes (C++) 6-6

Group 6: Enumerations 6-7

Group 7: Functions 6-8

Group 8: Constants 6-9

Group 9: Variables 6-10

Group 10: Name spaces (C++) 6-11

Group 11: Class templates (C++) 6-12

Group 12: Function templates (C++) 6-13

Group 20: Style 6-14

7 Code Metrics

8 Polyspace Report Components Alphabetical List

9 Con guration Parameters

Product mode 9-2
Settings 9-2
Dependency 9-2
Command-Line Information 9-2

Settings from (C) 9-3
Settings 9-3
Dependency 9-4
Command-Line Information 9-4

Settings from (C++) 9-5
Settings 9-5
Dependency 9-5
Command-Line Information 9-6

Use custom project le 9-7
Settings 9-7
Dependency 9-7
Command-Line Information 9-7

Project con guration 9-8
Settings 9-8
Dependency 9-8
Command-Line Information 9-8

Enable additional le list 9-9
Settings 9-9

Command-Line Information 9-9

Stub lookup tables 9-10
Settings 9-10
Tips 9-11
Command-Line Information 9-11

Input 9-12
Settings 9-12
Command-Line Information 9-12

Tunable parameters 9-13
Settings 9-13
Command-Line Information 9-13

Output 9-14
Settings 9-14
Command-Line Information 9-14

Model reference verification depth 9-15
Settings 9-15
Command-Line Information 9-15

Model by model verification 9-17
Settings 9-17
Command-Line Information 9-17

Output folder 9-18
Settings 9-18
Command-Line Information 9-18

Make output folder name unique by adding a suffix 9-19
Settings 9-19
Command-Line Information 9-19

Add results to current Simulink project 9-20
Settings 9-20
Dependencies 9-20
Command-Line Information 9-20

Open results automatically after verification 9-21
Settings 9-21
Command-Line Information 9-21

Check con guration before veri cation 9-22
Settings 9-22
Command-Line Information 9-22

Verify all occurrences 9-23
Settings 9-23
Command-Line Information 9-23

10 Approximations Used During Bug Finder Analysis

Inputs in Polyspace Bug Finder 10-2

Global Variables in Polyspace Bug Finder 10-3

Option Descriptions

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: In your project configuration, the option is on the `Target & Compiler` node. See “Dependencies” on page 1-3 for ways in which the source code language can be automatically determined.

Command line: Use the option `-lang`. See “Command-Line Information” on page 1-4.

Settings

Default: `C-CPP` for hand code and `C` for model-generated code

`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 argument to the `-lang` option.
  - If you do not specify the `-lang` option, the source code language is determined by whether your source files are compiled as C or C++ files.

`-lang` Argument	C or C++	Source Code Language
`c`		`C`
`cpp`		`CPP`
`cpp11`		`CPP` The option `C++11 extensions (-cpp11-extension)` is also enabled.
`auto` or no argument	C	`C`
`auto` or no argument	C++	`CPP`
`auto` or no argument	Both	`C-CPP`

Command-Line Information

Parameter: `-lang`
Value: `c | cpp | c-cpp`
Default: `c-cpp`
Example: `polyspace-bug-finder-nodesktop -lang c-cpp -sources "file1.c,file2.cpp"`
Example: `polyspace-bug-finder-nodesktop -lang c -sources "file1.c,file2.c"`

See Also

Topics

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

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: In your project configuration, the option is on the Target & Compiler node.

Command line: Use the option `-compiler`. See “Command-Line Information” on page 1-10.

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.

Settings

Default: `generic`

`generic`

Analysis allows only standard syntax.

For C code, syntax must follow the ANSI® C standard.

For C++ code, syntax must follow ISO®/IEC 14882:2003 C++ (C++ 2003). If you want to allow C++ 11 syntax (ISO/IEC 14882:2011 C++), also select `C++ 11 extensions`.

`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 more information, see “Limitations” on page 1-9.

`gnu4.8`

Analysis allows GCC 4.8 syntax.

For more information, see “Limitations” on page 1-9.

`gnu4.9`

Analysis allows GCC 4.9 syntax.

For more information, see “Limitations” on page 1-9.

`gnu5.x`

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

For more information, see “Limitations” on page 1-9.

`gnu6.x`

Analysis allows GCC 6.1, 6.2, and 6.3 syntax.

For more information, see “Limitations” on page 1-9.

`clang3.x`

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

`keil`

Analysis allows non-ANSI C syntax and semantics associated with the Keilo 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).

`codewarrior`

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

If you select `codewarrior`, the option `Target processor type (-target)` shows only the targets that are allowed for the NXP CodeWarrior compiler. See `NXP CodeWarrior Compiler (-compiler codewarrior)`

`diab`

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

If you select `diab`, 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`, 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`, 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)`.

`tasking`

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

If you select `tasking`, 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`, the option `Target processor type (-target)` shows only the targets that are allowed for the Texas Instruments compiler. See `Texas Instruments Compiler (-compiler ti)`.

Tips

- 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 enable `Check JSF 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);
    }
    ```
  - Forward declaration of function parameters.

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

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

- 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 | clang3.x | visual9.0 | visual10.0 | visual11.0 | visual12.0 | visual14.0 | keil | iar | codewarrior | diab | greenhills | iar-ew | tasking | ti`
Default: `generic`
Example: `polyspace-bug-finder-nodesktop -lang c -sources "file1.c,file2.c" -compiler gnu4.6`
Example: `polyspace-bug-finder-nodesktop -lang cpp -sources "file1.cpp,file2.cpp" -compiler visual9.0`

See Also

`Target processor type (-target)` | `C++11 extensions (-cpp11-extension)` | `Block char16/32_t types (-no-uliterals)`

Topics

“Supported Keil or IAR Language Extensions”
“Supported C++ 2011 Language Extensions”
“Troubleshooting in Polyspace Bug Finder”

Target processor type (-target)

Specify size of data types and endianness by using predefined target processor list

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: 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 1-15.

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]

Target	char	short	int	long	long long	float	double	long double[a]	ptr	Default sign of char	endian	Alignment
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 NPX CodeWarrior compiler	See NXP CodeWarrior Compiler (-compiler codewarrior)											
Targets for Diab compiler	See Diab Compiler (-compiler diab).											
Targets for Green Hills compiler	See Green Hills Compiler (-compiler greenhills).											
Targets for IAR Embedded Workbench compiler	See IAR Embedded Workbench Compiler (-compiler iar-ew)											
Targets for TASKING compiler	See TASKING Compiler (-compiler tasking).											

Target	char	short	int	long	long long	float	double	long double[a]	ptr	Default sign of char	endian	Alignment
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. If your target processor does not match the characteristics of a predefined processor, contact MathWorks® technical support.

Command-Line Information

Parameter: `-target`
Value: `i386 | sparc | m68k | powerpc | c-167 | x86_64 | tms320c3x | sharc21x61 | necv850 | hc08 | hc12 | mpc5xx | c18 | mcpu`
Default: `i386`
Example: `polyspace-bug-finder-nodesktop -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 Results
`Lower Estimate of Local Variable Size` | `Higher Estimate of Local Variable Size`

Topics

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

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, 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.

- In the user interface, add the folder containing your compiler headers to the project.

  For more information, see “Update Polyspace Project”.
- At the command line, use the flag `-I` with the `polyspace-bug-finder-nodesktop` command.

  For more information, see `-I`.

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the contextual help.

If you use the NXP CodeWarrior compiler flags to change any of these default specifications and want to emulate these flags, contact Technical Support.

Command-Line Information

Parameter: `-compiler codewarrior -target`
Value: `s12z | powerpc`
Default: `s12z`
Example: `polyspace-bug-finder-nodesktop -compiler codewarrior -target powerpc`

See Also

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

Topics

"Specify Target Environment and Compiler Behavior"

Introduced in R2018a

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, 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.

- In the user interface, add the folder containing your compiler headers to the project.

  For more information, see “Add Source Files for Analysis in Polyspace User Interface”.
- At the command line, use the flag `-I` with the `polyspace-bug-finder-nodesktop` command.

  For more information, see `-I`.

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 contextual help.

If you use Diab compiler flags to change any of these default specifications and want to emulate these flags, contact Technical Support.

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: `polyspace-bug-finder-nodesktop -compiler diab -target tricore`

See Also

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

Topics

“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, you see only the processors allowed for aGreen 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.

- In the user interface, add the folder containing your compiler headers to the project.

  For more information, see “Add Source Files for Analysis in Polyspace User Interface”.
- At the command line, use the flag `-I` with the `polyspace-bug-finder-nodesktop` command.

  For more information, see `-I`.

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the contextual help.

If you use the Green Hills compiler flags to change any of these default specifications and want to emulate these flags, contact Technical Support.

Tips

- 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: `polyspace-bug-finder-nodesktop -compiler greenhills -target arm`

See Also

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

Topics

“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, 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.

- In the user interface, add the folder containing your compiler headers to the project.

  For more information, see “Update Polyspace Project”.
- At the command line, use the flag `-I` with the `polyspace-bug-finder-nodesktop` command.

  For more information, see `-I`.

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the contextual help.

If you use the IAR Embedded Workbench compiler flags to change any of these default specifications and want to emulate these flags, contact Technical Support.

Tips

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

For the list of unsupported constructs, see `codeprover_limitations.pdf` in *`matlabroot`*`\polyspace\verifier\code_prover`. Here, *`matlabroot`* is the MATLAB® installation folder, for instance, `C:\Program Files\MATLAB\R2017b`.

Command-Line Information

Parameter: `-compiler iar-ew -target`
Value: `arm | avr | msp430 | rh850 | rl78`
Default: `arm`
Example: `polyspace-bug-finder-nodesktop -compiler iar-ew -target rl78`

See Also

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

Topics

"Specify Target Environment and Compiler Behavior"

Introduced in R2018a

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, 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.

- In the user interface, add the folder containing your compiler headers to the project.

  For more information, see “Add Source Files for Analysis in Polyspace User Interface”.
- At the command line, use the flag `-I` with the `polyspace-bug-finder-nodesktop` command.

  For more information, see `-I`.

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 contextual help.

If you use TASKING compiler flags to change any of these default specifications and want to emulate these flags, contact Technical Support.

Tips

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

  For the list of unsupported constructs, see `codeprover_limitations.pdf` in *`matlabroot`*`\polyspace\verifier\code_prover`. Here, *`matlabroot`* is the MATLAB installation folder, for instance, `C:\Program Files\MATLAB\R2017b`.
- 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: `polyspace-bug-finder-nodesktop -compiler tasking -target tricore`

See Also

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

Topics

“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, 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.

- In the user interface, add the folder containing your compiler headers to the project.

  For more information, see “Update Polyspace Project”.
- At the command line, use the flag `-I` with the `polyspace-bug-finder-nodesktop` command.

  For more information, see `-I`.

Settings

To see the default sizes in bits for the fundamental types that the targets use, see the contextual help.

If you use the Texas Instruments compiler flags to change any of these default specifications and want to emulate these flags, contact Technical Support.

Tips

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

For the list of unsupported constructs, see codeprover_limitations.pdf in *matlabroot*\polyspace\verifier\code_prover. Here, *matlabroot* is the MATLAB installation folder, for instance, C:\Program Files\MATLAB\R2017b.

Command-Line Information

Parameter: -compiler ti -target
Value: c28x | c6000 | arm | msp430
Default: c28x
Example: polyspace-bug-finder-nodesktop -compiler ti -target msp430

See Also

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

Topics

"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

The Generic target options dialog box opens when you set the Target processor type to mcpu.

Allows the specification of a generic “Micro Controller/Processor Unit" target. Use the dialog box to specify the name of a new mcpu target, for example *MyTarget*. That new target is added to the Target processor type option list.

Changing the genetic target has consequences for:

- Detection of overflow
- Computation of sizeof objects

The Target processor type option is available on the Target & Compiler node in the Con guration pane.

Settings

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

- char [8]
- short [16]
- int [16]
- long [32]
- long long [32]
- oat [32]
- double [32]
- long double [32]

- pointer `[16]`
- 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, specify your target with the other 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-nodesktop -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-nodesktop -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-nodesktop -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-nodesktop -target mcpu -char-is-16bits`

Option	Description	Available With	Example
`-short-is-8bits`	Define `short` as 8 bits, regardless of sign	`mcpu`	`polyspace-bug-finder-nodesktop -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-nodesktop -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-nodesktop -target mcpu -long-is-32bits`
`-long-long-is-64bits`	Define `long long` as 64 bits, regardless of sign. Alignment is also set to 64 bits.	`mcpu`	`polyspace-bug-finder-nodesktop -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-nodesktop -target mcpu -double-is-64bits`
`-pointer-is-24bits`	Define pointer as 24 bits, regardless of sign.	`c18`	`polyspace-bug-finder-nodesktop -target c18 -pointer-is-24bits`
`-pointer-is-32bits`	Define pointer as 32 bits, regardless of sign.	`mcpu`	`polyspace-bug-finder-nodesktop -target mcpu -pointer-is-32bits`

Option	Description	Available With	Example
`-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-nodesktop -target mcpu -align 16`

Common Generic Targets

The following tables describe the characteristics of common generic targets.

ST7 (Hiware C compiler : HiCross for ST7)

ST7	char	short	int	long	long long	oat	doubl e	long doubl e	ptr	char is	endia n
size	8	16	16	32	32	32	32	32	16/32	unsigne d	Big
alignment	8	16/8	16/8	32/16/8	32/16/8	32/16/8	32/16/ 8	32/16/ 8	32/16/ 8	N/A	N/A

ST9 (GNU C compiler : gcc9 for ST9)

ST9	char	short	int	long	long long	oat	double	long double	ptr	char is	endian
size	8	16	16	32	32	32	64	64	16/64	unsigned	Big
alignment	8	8	8	8	8	8	8	8	8	N/A	N/A

Hitachi H8/300, H8/300L

Hitachi H8/300, H8/300L	char	short	int	long	long long	oat	double	long double	ptr	char is	endian
size	8	16	16/32	32	64	32	654	64	16	unsigned	Big
alignment	8	16	16	16	16	16	16	16	16	N/A	N/A

Hitachi H8/300H, H8S, H8C, H8/Tiny

Hitachi H8/300H, H8S, H8C, H8/Tiny	char	short	int	long	long long	oat	double	long double	ptr	char is	endian
size	8	16	16/32	32	64	32	64	64	32	unsigned	Big
alignment	8	16	32/16	32/16	32/16	32/16	32/16	32/16	32/16	N/A	N/A

See Also

`Target processor type (-target)`

Topics

“Specify Target Environment and Compiler Behavior”

Respect C90 standard (-no-language-extensions)

Restrict analysis to C language specified in ANSI C standard

Description

Restrict the analysis to the C language specified in the ANSI C standard (ISO/IEC 9899:1990). Language extensions added in later standards such as C99 generate compilation errors.

Set Option

User interface: In your project configuration, the option is on the Target & Compiler node. See “Dependencies” on page 1-37 for other options you must also enable.

Command line: Use the option `-no-language-extensions`. See “Command-Line Information” on page 1-37.

Why Use This Option

Use this option if you compile your code by using the C90 standard.

For instance, if you compile with the GCC option `-ansi` or `-std=c90`, use this option.

Settings

☑ On

Restrict the analysis to the C90 standard. Code must conform to the ANSI C standard (ISO/IEC 9899:1990).

☐ Off (default)

Allow language extensions from the C99 standard (ISO/IEC 9899:1999).

Dependencies

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

If you enable this option, you cannot use `Compiler (-compiler)` settings `keil` and `iar`.

Command-Line Information

Parameter: `-no-language-extensions`
Default: off
Example: `polyspace-bug-finder-nodesktop -lang c -no-language-extensions`

See Also

Topics

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

Introduced in R2015b

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: In your project configuration, the option is on the Target & Compiler node. See “Dependency” on page 1-38 for other options you must also enable.

Command line: Use the option `-sfr-types`. See “Command-Line Information” on page 1-39.

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: `polyspace-bug-finder-nodesktop -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: 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 1-41.

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 de ned, 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: `polyspace-bug-finder-nodesktop -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: 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 1-44.

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, compile this code using the compiler settings that you typically use:

```
#include <assert.h>
#include <limits.h>

enum { MAXSIGNEDBYTE=127 } mysmallenum_t;
int dummy[(int)sizeof(mysmallenum_t) - (int)sizeof(int)]; /* Breakpoint 1 */

enum { MYMAXINT = INT_MAX } myintenum_t;
int main(void) {
```

```
    assert((MYMAXINT + 1) < 0);  /* Breakpoint 2 */
    assert((MYMAXINT + 1) >= 0); /* Breakpoint 3 */
    assert(0);  /* Breakpoint 4 */

    return 0;
}
```

If compilation does not fail even at breakpoint 4, your `assert` statements do not behave as expected. Check your compiler documentation and change your compiler settings. If compilation fails at:

- Breakpoint 1: Use `defined-by-compiler` for this option.
- Breakpoint 2: Use `auto-signed-first` for this option.
- Breakpoint 3: Use `auto-unsigned-first` for this option.

Settings

Default: `defined-by-compiler`

`defined-by-compiler`

Uses the signed integer type for all compilers except gnu.

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

`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 isnt`, `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: `polyspace-bug-finder-nodesktop -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 1-46.

Set Option

User interface: 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 1-46.

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: `polyspace-bug-finder-nodesktop -logical-signed-right-shift`

See Also

Topics

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

C++11 extensions (-cpp11-extension)

Allow C++11 language extensions

Description

Allow C++11 language extensions.

Set Option

User interface: In your project configuration, the option is on the Target & Compiler node. See “Dependencies” on page 1-48 for other options you must also enable.

Command line: Use the option `-cpp11-extension`. See “Command-Line Information” on page 1-48.

Why Use This Option

If your compiler allows C++11 language extensions, enable this option.

To check if your compiler allows the extensions, compile this code using the compiler settings that you typically use:

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

If the code compiles, enable this option.

For instance, on a GCC compiler, the code compiles with the `-std=c++11` flag but fails to compile without the flag. If you typically use the flag, enable this option.

Settings

☑ On

The analysis allows C++11 syntax.

☐ Off (default)

The analysis does not allow C++11 syntax.

Dependencies

This analysis option is available only when both these conditions are true:

- `Source code language (-lang)` is `CPP` or `C-CPP`.
- `Compiler (-compiler)` is `generic`, `gnu4.6`, or `gnu4.7`.

Command-Line Information

Parameter: `-cpp11-extension`
Default: off
Example: `polyspace-bug-finder-nodesktop -lang cpp -cpp11-extension`

See Also

`Compiler (-compiler)` | `Block char16/32_t types (-no-uliterals)`

Topics

“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”
“Supported C++ 2011 Language Extensions”

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: 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 `CPP` or `C-CPP`.
- `Compiler (-compiler)` is either `generic` or a `gnu` version.

Command-Line Information

Parameter: `-no-uliterals`
Default: off
Example: `polyspace-bug-finder-nodesktop -lang cpp -compiler gnu4.7 -cpp11-extension -no-uliterals`

See Also

`Compiler (-compiler)` | `C++11 extensions (-cpp11-extension)`

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: 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. See “#pragma Directives” (Polyspace Code Prover).

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: `polyspace-bug-finder-nodesktop -compiler visual10 -pack-alignment-value 4`

See Also

Topics

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

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: 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. See also “#pragma Directives” (Polyspace Code Prover).

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: `polyspace-bug-finder-nodesktop -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 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: 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 options cause `size_t` to be defined as `unsigned long` or `unsigned long long`. Replace `unsigned int` with `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-int` | `unsigned-long` | `unsigned-long-long`
Default: `defined-by-compiler`
Example: `polyspace-bug-finder-nodesktop -size-t-type-is unsigned-long`

See Also

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: 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 1-58.

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: `polyspace-bug-finder-nodesktop -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: In your project configuration, the option is on the Environment Settings node. See “Dependency” on page 1-60 for other options that you must also enable.

Command line: Use the option -no-extern-c. See “Command-Line Information” on page 1-60.

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: `polyspace-bug-finder-nodesktop -lang cpp -no-extern-C`

See Also

Topics

"Specify Polyspace Analysis Options"

Preprocessor de nitions (-D)

Replace macros in preprocessed code

Description

Replace macros with their definitions in preprocessed code.

Set Option

User interface: In your project configuration, the option is on the Macros node.

Command line: Use the option -D. See “Command-Line Information” on page 1-63.

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, 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`.

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: `polyspace-bug-finder-nodesktop -D HAVE_MYLIB -D int32_t=int`

See Also

`Disabled preprocessor definitions (-U)`

Topics

“Specify Polyspace Analysis Options”

Disabled preprocessor de nitions (-U)

Undefine macros in preprocessed code

Description

Undefine macros in preprocessed code.

Set Option

User interface: In your project configuration, the option is on the Macros node.

Command line: Use the option -U. See “Command-Line Information” on page 1-65.

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.

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: `polyspace-bug-finder-nodesktop -U HAVE_MYLIB -U USE_COM1`

See Also

`Preprocessor definitions (-D)`

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: 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"
```

Off

Characters are not controlled for files names or paths.

Command-Line Information

Parameter: `-dos`
Default: Off
Example: `polyspace-bug-finder-nodesktop -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: 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 1-69.

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.

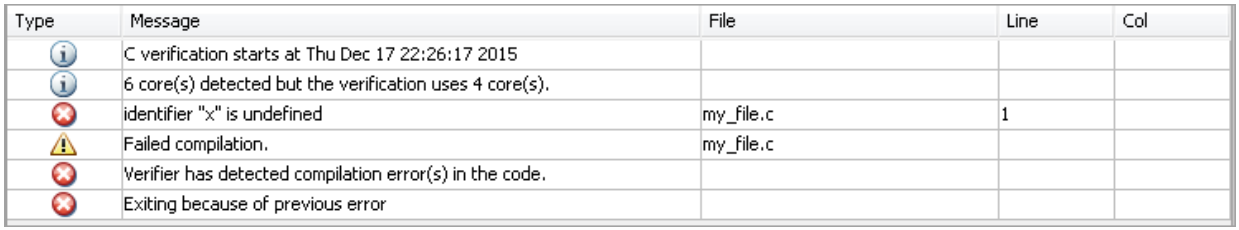
The option is more useful for a Code Prover analysis because the Code Prover run-time checks rely more heavily on range propagation across functions.

Settings

☑ On

The analysis stops even if a single compilation error occurs.

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 and Linking Errors" (Polyspace Code Prover).

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, 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.

Command-Line Information

Parameter: `-stop-if-compile-error`
Default: Off
Example: `polyspace-bug-finder-nodesktop -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: 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 1-73.

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 -function-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. After the verification, this script is executed.

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 *`matlabroot`*`\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 *`matlabroot`*`\sys\perl\win32\bin\perl.exe <`*`absolute_path`*`>\replace_keyword.pl`.

  Here, *`matlabroot`* is the location of the current MATLAB installation such as `C:\Program Files\MATLAB\R2015b\` 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
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®: `polyspace-bug-finder-nodesktop -sources file_name -post-preprocessing-command `pwd`/replace_keyword.pl`
Example in Windows: `polyspace-bug-finder-nodesktop -sources file_name -post-preprocessing-command "C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\replace_keyword.pl"`

See Also

`Command/script to apply after the end of the code verification (-post-analysis-command)`

Topics

“Specify Polyspace Analysis Options”

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: In your project configuration, the option is on the Environment Settings node.

Command line: Use the option -include. See “Command-Line Information” on page 1-76.

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: `polyspace-bug-finder-nodesktop -include `pwd`/sources/a_file.h -include /inc/inc_file.h`

See Also

Topics

“Gather Compilation Options Efficiently”
“Specify Polyspace Analysis Options”

Include folders (-I)

View include folders used for analysis

Description

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 > Con guration. 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. Unlike other options, you do not specify include folders on the Con guration pane. Instead, you add your include folders on the Project Browser pane.

Command-Line Information

Parameter: -I
Value: Folder name
Example: polyspace-bug-finder-nodesktop -I /com1/inc -I /com1/sys/inc

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: 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 1-80.

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.

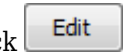
Because of these broad assumptions, Polyspace can sometimes produce false positives.

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: `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-data-range-specifications "C:\DRS\range.xml"`

See Also

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

Topics

“Specify Polyspace Analysis Options”
“Specify External Constraints”

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

Consider global variables as uninitialized

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.

Set Option

User interface: 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 1-82.

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.

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

- If you initialize a global variable using the generated `main`:
  - Polyspace does not produce a red `Non-initialized variable` error if your code reads the variable before writing to it. The error is not produced even if you turn on the option `Ignore default initialization of global variables`.
  - Polyspace considers that before the first write operation on the variable in a function, the variable can take any value allowed by its type.

  For more information on initializing global variables using the generated `main`, see `Variables to initialize (-main-generator-writes-variables)`.
- 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

See Also

Topics

“Specify Polyspace Analysis Options”

No STL stubs (-no-stl-stubs)

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

Description

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

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

Set Option

User interface: In your project configuration, the option is on the Inputs & Stubbing node. See “Dependency” on page 1-84 for other options that you must also enable.

Command line: Use the option `-no-stl-stubs`. See “Command-Line Information” on page 1-84.

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

See Also

Functions to stub (-functions-to-stub)

Specify functions to stub during analysis

Description

This option a ects a Code Prover analysis only.

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: 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 1-87.

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)`

Function Type	Basic Syntax	Argument Syntax
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)`

Command-Line Information

Parameter: `-functions-to-stub`
No Default
Value: *`function1`*`[,`*`function2`*`[,...]]`
Example: `polyspace-bug-finder-nodesktop -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: 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 1-90.

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 `-function-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: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-stub-embedded-coder-lookup-table-functions`

See Also

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.

Set Option

User interface: 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 1-93.

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.

Generate results for sources and	Do not generate results for	Final Result
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: `all-headers` | `custom=`*`file1`*`[,`*`file2`*`[,...]]` | *`folder1`*`[,`*`folder2`*`[,...]]`
Example: `polyspace-bug-finder-nodesktop -lang c -sources` *`file_name`* `-misra2 required-rules -generate-results-for "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.

Set Option

User interface: 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 1-98.

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).

`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.

Generate results for sources and	Do not generate results for	Final Result
`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 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` | `custom=`*`file1`*`[,`*`file2`*`[,...]]` | *`folder1`*`[,`*`folder2`*`[,...]]`
Example: `polyspace-bug-finder-nodesktop -lang c -sources` *`file_name`* `-misra2 required-rules -do-not-generate-results-for "C:\usr\include"`

See Also

Topics

“Specify Polyspace Analysis Options”

Introduced in R2016a

External multitasking con guration

Enable setup of multitasking configuration from external file definitions

Description

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 Con guration pane, the option is available on the Multitasking node.

Command line: See “Command-Line Information” on page 1-100.

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 mulitasking 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”
“Configuring Polyspace Multitasking Analysis Manually”

Introduced in R2018a

OIL les selection (-osek-multitasking)

Set up multitasking configuration from OIL file definition

Description

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

Set Option

User interface: In the Con guration pane, the option is available on the Multitasking node. See Dependencies on page 1-104 for other options you must also enable.

Command line: Use the option -osek-multitasking. See "Command-Line Information" on page 1-105.

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 projet and run an analysis on this project. To try the steps in this example, use the demo files in the folder *`MATLABROOT`*`/help/toolbox/bugfinder/examples/External_multitasking/OSEK`. *`MATLABROOT`* is your MATLAB 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 *MATLABROOT*/help/toolbox/bugfinder/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:

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

Polyspace detects a data race on page 3-93 on variable `var3` because of multiple read and write operation from tasks `init` and `afterinit1`.

```
#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.

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, first select the option `External multitasking configuration`.

Command-Line Information

Parameter: `-osek-multitasking`
Value: `auto` | `custom='`*`file1 [,file2, dir1,...]`*`'`
Default: Off
Example: `polyspace-bug-finder-nodesktop -sources` *`source_path`* `-I` *`include_path`* `-osek-multitasking custom='path\to\file1.oil, path\to\dir'`

Related Examples

- “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.

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 1-107 for other options you must also enable.

Command line: Use the option `-autosar-multitasking`. See "Command-Line Information" on page 1-105.

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, first select the option External multitasking configuration.

Command-Line Information

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

Related Examples

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

Introduced in R2018a

Con gure multitasking manually

Consider that code is intended for multitasking

Description

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

Set Option

User interface: In your project configuration, the option is available on the Multitasking node.

Command line: See “Command-Line Information” on page 1-110.

Why Use This Option

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.

The 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.

☐ Off (default)

The code is not intended for a multitasking application.

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 the `-entry-points` option, you turn on multitasking analysis.

See Also

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

Topics

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

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.

Specify whether to use the automatic concurrency detection for POSIX®, VxWorks®, Windows, and µC/OS II multithreading functions.

Set Option

User interface: 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 1-115.

Why Use This Option

If you use this option, Polyspace determines your multitasking model from your use of multithreading functions.

In some cases, using automatic concurrency detection can slow down the Code Prover analysis. In those cases, you can explicitly specify your multitasking model using the option Configure multitasking manually.

Settings

☑ On

If you use POSIX, VxWorks, Windows, or µC/OS II functions for multitasking, the analysis automatically detects your multitasking model from your code.

The supported multitasking functions are the following:

Family	Thread Creation	Critical Section Begins	Critical Section Ends
POSIX	`pthread_create`	`pthread_mutex_lock`	`pthread_mutex_unlock`
VxWorks	`taskSpawn`	`semTake`	`semGive`
Windows	`CreateThread`	`EnterCriticalSection`	`LeaveCriticalSection`
µC/OS II	`OSTaskCreate`	`OSMutexPend`	`OSMutexPost`

To activate automatic detection of concurrency primitives for VxWorks, use the `VxWorks` template. For more information on templates, see “Create Project Using Configuration Template”.

`Note` For VxWorks, concurrency detection is possible only if the multitasking functions are created from an entry point named `main`. If the entry point has a different name, such as `vxworks_entry_point`, do the following:

1. `Verify whole application`: Select verify the whole application.
2. `Preprocessor definitions (-D)`: In preprocessor definitions, set `vxworks_entry_point=main`.
3. `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`: Enable automatic concurrency detection.

☐ 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 `Configure multitasking manually`.

Dependencies

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

Limitations

The multitasking model extracted by Polyspace does not include some features. Polyspace cannot model:

- Thread priorities and attributes — Ignored by Polyspace
- Recursive semaphores
- Unbounded thread identifiers — Warning

  For example:

  ```
  extern pthread_t ids[]
  ```

  Or

  ```
  pthread_t* ids = (pthread_t* malloc(n*sizeof(pthread_t))
  ```
- Calls to concurrency primitive through high-order calls — Warning.
- Termination of threads — Polyspace ignores `pthread_join`, and replaces `pthread_exit` by a standard exit.
- Shared local variables — Only global variables are considered shared. If a local variable is accessed by multiple threads, the analysis does not take into account the shared nature of the variable.

  Example

  In this example, the analysis does not take into account that the local variable `x` can be accessed by both `task1` and `task2` (after the new thread is created).

  ```
  #include <pthread.h>
  #include <stdlib.h>


  void* task2(void* args) {
      int* x = (int*) args;
  ```

```
    *x = 1;
    return (void*)x;
}

void task1() {
    int x;
    x = 2;
    pthread_t id;
    (void)pthread_create(&id,NULL,task2,(void*) &x);
                      /* x (local var) passed to task2 */
    x = 3 ;

    /* Unknown thread priority means x = 1 OR x = 3.*/
    /* However, the analysis considers  x = 3 */
    /* Assertion below is green */
    assert(x==3);
}

int main(void) {
    task1();
    return 0;
}
```

- Shared dynamic memory — Only global variables are considered shared. If a dynamically allocated memory region is accessed by multiple threads, the analysis does not take into account its shared nature.

  Example

  In this example, the analysis does not take into account that `lx` points to a shared memory region. The region can be accessed by both `task1` and `task2` (after the new thread is created). The Code Prover analysis also reports `lx` as a non-shared variable.

  ```
  #include <pthread.h>
  #include <stdlib.h>

  static int *lx;

  void* task2(void* args) {
      int* x = (int*) args;
      *x = 1;
      return (void*)x;
  }

  void task1() {
  ```

```
    pthread_t id;
    lx = (int *)malloc(sizeof(int));

    if(lx==NULL) exit(1);

    (void)pthread_create(&id,NULL,task2,(void*) lx);

    *lx = 3 ;

    /* Unknown thread priority means *lx = 1 OR *lx = 3.*/
    /* However, the analysis considers  *lx = 3 */
    /* Assertion below is green */
    assert(*lx==3);
}

int main(void) {
    task1();
    return 0;
}
```

Command-Line Information

Parameter: `-enable-concurrency-detection`
Default: Off
Example: `polyspace-code-prover-nodesktop -sources file_name -enable-concurrency-detection`

See Also

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

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

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: 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 `-entry-points`. See “Command-Line Information” on page 1-117.

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.

The 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, first select the option `Configure multitasking manually`.

Tips

If you specify a function as a task, you must provide its definition. Otherwise, the analysis does not consider the function as a task.

Command-Line Information

Parameter: `-entry-points`
No Default
Value: *`function1`*`[,`*`function2`*`[,...]]`
Example: `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-entry-points func_1,func_2`

See Also

`Critical section details (-critical-section-begin -critical-section-end)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics

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

Cyclic tasks (-cyclic-tasks)

Specify functions that represent cyclic tasks

Description

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 on page 1-116, other cyclic tasks and interrupts on page 1-121.

  To model a cyclic task that cannot be interrupted by other cyclic tasks, specify the task as nonpreemptable. See `-non-preemptable-tasks`.

Set Option

User interface: In your project configuration, the option is available on the Multitasking node. See "Dependencies" on page 1-119 for other options you must also enable.

Command line: Use the option `-cyclic-tasks`. See "Command-Line Information" on page 1-119.

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);
```

The 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 `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, first select the option `Configure multitasking manually`.

Command-Line Information

Parameter: `-cyclic-tasks`
No Default

Value: *function1*[,*function2*[,...]]
Example: `polyspace-bug-finder-nodesktop -sources file_name -cyclic-tasks func_1,func_2`

See Also

`-preemptable-interrupts` | `-non-preemptable-tasks` | `Interrupts (-interrupts)` | `Tasks (-entry-points)` | `Critical section details (-critical-section-begin -critical-section-end)` | `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"

Introduced in R2016b

Interrupts (-interrupts)

Specify functions that represent nonpreemptable interrupts

Description

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 on page 1-116, cyclic tasks on page 1-118 or other interrupts.

  To model an interrupt that can be interrupted by other interrupts, specify the interrupt as preemptable. See `-preemptable-interrupts`.

Set Option

User interface: 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 `-interrupts`. See "Command-Line Information" on page 1-123.

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);
```

The 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, first select the option `Configure multitasking manually`.

Tips

If you specify a function as an interrupt, you must provide its definition. Otherwise, the analysis does not consider the function as an interrupt.

Command-Line Information

Parameter: -interrupts
No Default
Value: *function1*[,*function2*[,...]]
Example: polyspace-bug-finder-nodesktop -sources *file_name* -interrupts func_1,func_2

See Also

-preemptable-interrupts | -non-preemptable-tasks | Cyclic tasks (-cyclic-tasks) | Tasks (-entry-points) | Critical section details (-critical-section-begin -critical-section-end) | 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"

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.

Specify a routine that disables all interrupts and a routine that reenables all interrupts.

Set Option

User interface: In your project configuration, the option is available on the Multitasking node. See “Dependencies” on page 1-125 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 1-126.

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, you must select the option, `Configure multitasking manually`.

Tips

- The routine that you specify for the option disables preemption by all:
  - Noncylic Tasks on page 1-116
  - Cyclic tasks on page 1-118
  - Interrupts on page 1-121

  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.

Command-Line Information

Parameter: `-routine-disable-interrupts | -routine-enable-interrupts`
No Default
Value: *`function_name`*
Example: `polyspace-bug_finder-nodesktop -sources` *`file_name`* `-routine-disable-interrupts atomic_section_begins -routine-enable-interrupts atomic_section_ends`

See Also

`Configure multitasking manually` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)` | `Data race` | `Data race including atomic operations`

Topics

"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"

Introduced in R2017a

Critical section details (-critical-section-begin -critical-section-end)

Specify functions that begin and end critical sections

Description

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: In your project configuration, the option is available on the `Multitasking` node. See “Dependencies” on page 1-129 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 1-130.

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();
}
```

The analysis uses the critical section information to look for concurrency defects such as data race and deadlock. See “Concurrency Defects”.

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, first select the option Configure multitasking manually.

Tips

- 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: `func_begin`	Starting routine: `func_begin`
Ending routine: `func_end`	Ending routine: `func_end`

`void my_task1() {`
`  my_lock(1);`
`  /* Critical section code */`
`  my_unlock(1);`
`}`
	`void my_task2() {`
	`  my_lock(2);`
	`  /* Critical section code */`
	`  my_unlock(2);`
	`}`

- 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.

Command-Line Information

Parameter: `-critical-section-begin` | `-critical-section-end`
No Default
Value: *`function1`*`:cs1[,`*`function2`*`:cs2[,...]]`
Example: `polyspace-bug_finder-nodesktop -sources` *`file_name`* `-critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`

See Also

`Configure multitasking manually` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)` | `Data race` | `Data race including atomic operations`

Topics

“Specify Polyspace Analysis Options”

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

Temporally exclusive tasks (-temporal-exclusions-file)

Specify entry point functions that cannot execute concurrently

Description

Specify entry point functions that cannot execute concurrently. The execution of the functions cannot overlap with each other.

Set Option

User interface: In your project configuration, the option is available on the Multitasking node. See “Dependencies” on page 1-133 for other options you must also enable.

Command line: Use the option -temporal-exclusions-file. See “Command-Line Information” on page 1-133.

Why Use This Option

Use this option to implement temporal exclusion in multitasking code.

The analysis uses the temporal exclusion information to look for concurrency defects such as data race. See Data race.

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, first select the option `Configure multitasking manually`.

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.

Parameter: `-temporal-exclusions-file`
No Default
Value: Name of temporal exclusions file
Example: `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-temporal-exclusions-file "C:\exclusions_file.txt"`

See Also

`Configure multitasking manually` | `Tasks (-entry-points)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Data race` | `Data race including atomic operations`

Topics

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

Check MISRA C:2004 (-misra2)

Check for violation of MISRA C:2004 rules

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: In your project configuration, the option is on the Coding Rules & Code Metrics node. See “Dependencies” on page 1-136 for other options that you must also enable.

Command line: Use the option `-misra2`. See “Command-Line Information” on page 1-136.

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 rule 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.

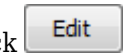
`SQO-subset1`

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:2004)".

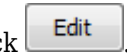
`SQO-subset2`

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)".

`custom`

Specify coding rules to check. Click 

to create a coding rules file. After creating and saving the file, to reuse it for another project, do one of the following:

- Enter full path to the file in the space provided.
- Click 

. Click to load the file.

Format of the custom file:

```
rule number off|on
```

Use # to enter comments in the file. For example:

```
10.5 off # rule 10.5: type conversion
17.2 on # rule 17.2: pointers
```

`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.

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` | *`file`*
Default: `required-rules`
Example: `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-misra2 all-rules`

See Also

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

Topics

“Specify Polyspace Analysis Options”
“Check for Coding Rule Violations”
“Polyspace MISRA C 2004 and MISRA AC AGC Checkers”
“Software Quality Objective Subsets (C:2004)”

Check MISRA AC AGC (-misra-ac-agc)

Check for violation of MISRA AC AGC rules

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: In your project configuration, the option is on the Coding Rules & Code Metrics node. See “Dependencies” on page 1-140 for other options that you must also enable.

Command line: Use the option `-misra-ac-agc`. See “Command-Line Information” on page 1-140.

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 rule 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.

`all-rules`

Check required, recommended and readability-related rules.

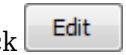
`SQO-subset1`

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`

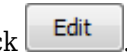
Check a subset of rules including `SQO-subset1` and some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see “Software Quality Objective Subsets (AC AGC)”.

`custom`

Specify coding rules to check. Click 

to create a coding rules file.

After creating and saving the file, to reuse it for another project, do one of the following:

- Enter full path to the file in the space provided.
- Click 

. Click to load the file.

Format of the custom file:

```
rule number off|on
```

Use # to enter comments in the file. For example:

```
10.5 off # rule 10.5: type conversion
17.2 on # rule 17.2: pointers
```

`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.

This setting is not available from the drop-down list in the user interface. To choose this setting, enter the option `-misra-ac-agc single-unit-rules` in the field `Other`.

`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.

This setting is not available from the drop-down list in the user interface. To choose this setting, enter the option `-misra-ac-agc system-decidable-rules` in the field `Other`.

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` | `all-rules` | `SQO-subset1` | `SQO-subset2` | `single-unit-rules` | `system-decidable-rules` | *`file`*
Default: `OBL-rules`
Example: `polyspace-bug-finder-nodesktop -sources file_name -misra-ac-agc all-rules`

See Also

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

Topics

“Specify Polyspace Analysis Options”
“Check for Coding Rule Violations”
“Polyspace MISRA C 2004 and MISRA AC AGC Checkers”
“MISRA C:2004 and MISRA AC AGC Coding Rules”
“Software Quality Objective Subsets (AC AGC)”

Check MISRA C:2012 (-misra3)

Check for violations of MISRA C:2012 rules and directives

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: In your project configuration, the option is on the Coding Rules & Code Metrics node. See “Dependencies” on page 1-144 for other options that you must also enable.

Command line: Use the option `-misra3`. See “Command-Line Information” on page 1-145.

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 rule violations. On the Source pane, for every coding rule violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

Settings

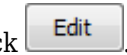
Default: `mandatory-required`

`mandatory-required`

Check for mandatory and required guidelines.

- Mandatory guidelines: Your code must comply with these guidelines.
- Required guidelines: You may deviate from these 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 

. 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.

`mandatory`

Check for mandatory guidelines.

`CERT-rules`

Check for a subset of coding rules that corresponds to CERT-C rules.

See “CERT C Coding Standard and Polyspace Results”.

`CERT-all`

Check for a subset of coding rules that corresponds to CERT-C rules and recommendations.

See “CERT C Coding Standard and Polyspace Results”.

`ISO-17961`

Check for a subset of coding rules that corresponds to the ISO/IEC TS 17961 coding standard.

`all`

Check for mandatory, required, and advisory guidelines.

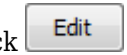
`SQO-subset1`

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)”.

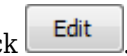
`SQO-subset2`

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)”.

`custom`

Specify guidelines to check. Click 

to create a coding rules file. Save the file. To reuse it for another project, do one of the following:

- Enter full path to the file in the space provided.
- Click 

. Click to load the file.

Custom file format:

```
rule number off|on
```

Use # to enter comments in the file. For example:

```
10.5 off # rule 10.5: essential type model
17.2 on # rule 17.2: functions
```

`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.

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` | `CERT-rules` | `CERT-all` | `ISO-17961` | `all` | `SQO-subset1` | `SQO-subset2` | `single-unit-rules` | `system-decidable-rules` | *`file`*
Default: `mandatory-required`
Example: `polyspace-bug-finder-nodesktop -lang c -sources` *`file_name`* `-misra3 mandatory-required`

See Also

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

Topics

"Specify Polyspace Analysis Options"

“Check for Coding Rule Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

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: In your project configuration, the option is on the Coding Rules & Code Metrics node. See “Dependency” on page 1-148 for other options that you must also enable.

Command line: Use the option -misra3-agc-mode. See “Command-Line Information” on page 1-149.

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
- 14.4, 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: `polyspace-bug-finder-nodesktop -sources file_name -misra3 all -misra3-agc-mode`

See Also

`Generate results for sources and (-generate-results-for)` | `Check MISRA C:2012 (-misra3)`

Topics

“Specify Polyspace Analysis Options”
“Check for Coding Rule Violations”
“Polyspace MISRA C:2012 Checkers”

Check custom rules (-custom-rules)

Follow naming conventions for identifiers

Description

Define naming conventions for identifiers and check your code against them.

Set Option

User interface: In your project configuration, the option is on the Coding Rules & Code Metrics node.

Command line: Use the option -custom-rules. See “Command-Line Information” on page 1-153.

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.

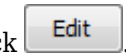
After analysis, the Results List pane lists violations of the naming conventions. On the Source pane, for every violation, Polyspace assigns a symbol to the keyword or identifier relevant to the violation.

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 

. The New File window opens.
2. From the drop-down list Set the following state to all Custom C, select Off. Click Apply.
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 elds must follow the speci ed 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 elds must follow the speci ed 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

Expression	Meaning
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.

- Manually edit an existing custom coding rules file:

  1. Open the file with a text editor.
  2. For every custom rule you want to check, enter the following information in adjacent lines.

     a. Rule number, followed by `on`. For example:

     ```
     4.3 on
     ```

     b. The error message you want to display starting with `convention=`. For example:

     ```
     convention=All struct fields must begin with s_
     ```

     c. The text pattern starting with `pattern=`. For example:

     ```
     pattern=s_[A-Za-z0-9_]
     ```

To use an existing coding rules file, enter the full path to the file in the field provided or use in the New File window to navigate to the file location.

Off (default)

Polyspace does not check your code against custom naming conventions.

Command-Line Information

Parameter: `-custom-rules`
Value: Name of coding rules file
Default: Off
Example: `polyspace-bug-finder-nodesktop -sources file_name -custom-rules "C:\Rules\myrules.txt"`

See Also

Topics

“Specify Polyspace Analysis Options”
“Create Custom Coding Rules”
“Format of Custom Coding Rules File”

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 only if you have defined it through a `typedef` statement in your source code.

Set Option

User interface: In your project configuration, the option is on the Coding Rules & Code Metrics node. See “Dependencies” on page 1-156 for other options that you must also enable.

Command line: Use the option `-boolean-types`. See “Command-Line Information” on page 1-156.

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 on page 5-160	Operands shall not be of an inappropriate essential type
10.3 on page 5-169	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 on page 5-173	The value of an expression should not be cast to an inappropriate essential type
14.4 on page 5-247	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.
16.7 on page 5-286	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 available only if you select `Check MISRA AC AGC (-misra-ac-agc)`, `Check MISRA C:2004 (-misra2)`, or `Check MISRA C:2012 (-misra3)`.

Command-Line Information

Parameter: `-boolean-types`
Value: *`type1`*`[,`*`type2`*`[,...]]`
No Default
Example: `polyspace-bug-finder-nodesktop -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 Rule Violations"

Allowed pragmas (-allowed-pragmas)

Specify pragma directives for which MISRA C:2004 rule 3.4 must not be applied

Description

Specify pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ rule 16-6-1 must not be applied.

Set Option

User interface: In your project configuration, the option is on the Coding Rules & Code Metrics node. See “Dependencies” on page 1-158 for other options that you must also enable.

Command line: Use the option -allowed-pragmas. See “Command-Line Information” on page 1-158.

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 the following options:

- `Check MISRA C:2004 (-misra2)`
- `Check MISRA AC AGC (-misra-ac-agc).`
- `Check MISRA C++ rules (-misra-cpp)`

Command-Line Information

Parameter: `-allowed-pragmas`
Value: *`pragma1`*`[,`*`pragma2`*`[,...]]`
No Default
Example: `polyspace-bug-finder-nodesktop -sources` *`filename`* `-misra-cpp required-rules -allowed-pragmas pragma_01,pragma_02`
Example: `polyspace-bug-finder-nodesktop -sources` *`filename`* `-misra2 required-rules -allowed-pragmas pragma_01,pragma_02`

See Also

`Check MISRA C:2004 (-misra2)|Check MISRA AC AGC (-misra-ac-agc)|Check MISRA C++ rules (-misra-cpp)`

Topics

“Check for Coding Rule Violations”
“MISRA C:2004 and MISRA AC AGC Coding Rules”
“MISRA C++ Coding Rules”

Check MISRA C++ rules (-misra-cpp)

Check for violations of MISRA C++ rules

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: In your project configuration, the option is on the Coding Rules & Code Metrics node. See “Dependency” on page 1-160 for other options that you must also enable.

Command line: Use the option -misra-cpp. See “Command-Line Information” on page 1-161.

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 rule 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.

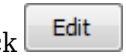
`SQO-subset1`

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`

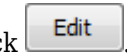
Check a subset of rules including `SQO-subset1` and some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see "Software Quality Objective Subsets (C++)"

`custom`

Specify coding rules to check. Click 

to create a coding rules file.

After creating and saving the file, to reuse it for another project, do one of the following:

- Enter full path to the file in the space provided.
- Click 

. Click to load the file.

Format of the custom file:

```
<rule number> off|on
```

Use # to enter comments in the file. For example:

```
9-5-1 off # rule 9-5-1: classes
15-0-2 on # rule 15-0-2: exception handling
```

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 | *file*
Default: required-rules
Example: polyspace-bug-finder-nodesktop -sources *file_name* -misra-cpp all-rules

See Also

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

Topics

“Specify Polyspace Analysis Options”
“Check for Coding Rule Violations”
“Polyspace MISRA C++ Checkers”
“Software Quality Objective Subsets (C++)”
“MISRA C++ Coding Rules”

Check JSF C++ rules (-jsf-coding-rules)

Check for violations of JSF C++ rules

Description

Specify whether to check for violation of JSF C++ rules (JSF++:2005). Each value of the option corresponds to a subset of rules to check.

Set Option

User interface: In your project configuration, the option is on the Coding Rules & Code Metrics node. See “Dependency” on page 1-163 for other options that you must also enable.

Command line: Use the option `-jsf-coding-rules`. See “Command-Line Information” on page 1-164.

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 rule 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.

`custom`

Specify coding rules to check. Click Edit to create a coding rules file.

After creating and saving the file, to reuse it for another project, do one of the following:

- Enter full path to the file in the space provided.
- Click Edit. Click to load the file.

Format of the custom file:

```
<rule number> off|on
```

Use # to enter comments in the file. For example:

```
67 off # rule 67: classes
202 on # rule 202: expressions
```

Tips

- If your project uses a setting other than `iso` 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` | *`file`*
Default: `shall-rules`
Example: `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-jsf-coding-rules all-rules`

See Also

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

Topics

“Specify Polyspace Analysis Options”
“Check for Coding Rule Violations”
“Polyspace JSF C++ Checkers”
“JSF C++ Coding 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: In your project configuration, the option is on the Coding Rules & Code Metrics node.

Command line: Use the option -code-metrics. See “Command-Line Information” on page 1-166.

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 .

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)`.

Command-Line Information

Parameter: `-code-metrics`
Default: Off
Example: `polyspace-bug-finder-nodesktop -sources file_name -code-metrics`

See Also

Topics

“Compute Code Complexity Metrics”

Find defects (-checkers)

Enable or disable defect checkers

Description

This option affects Bug Finder analysis only.

Enable checkers for bugs/coding defects.

Set Option

User interface: 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 1-168.

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. For information on which defects are default, refer to the individual defect reference pages.

`all`

All defects.

`CWE`

A subset of defects that correspond to CWE® IDs.

See “CWE Coding Standard and Polyspace Results”.

`CERT-rules`

A subset of defects that corresponds to CERT-C rules.

See “CERT C Coding Standard and Polyspace Results”.

`CERT-all`

A subset of defects that corresponds to CERT-C rules and recommendations.

See “CERT C Coding Standard and Polyspace Results”.

`ISO-17961`

A subset of defects that corresponds to ISO/IEC TS 17961 coding standard.

See “ISO/IEC TS 17961 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 *`matlabroot`*`\polyspace\resources`. Here, *`matlabroot`* is the MATLAB installation folder, such as `C:\Program Files\MATLAB\R2017a`.

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 | CWE | CERT-rules | CERT-all | ISO-17961 | defect group | defect parameters`
Default: `default`
Parameter: `-disable-checkers`
Value: `defect group | defect parameters`
Example 1: `polyspace-bug-finder-nodesktop -sources filename -checkers numerical,dataflow -disable-checkers FLOAT_ZERO_DIV`
Example 2: `polyspace-bug-finder-nodesktop -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"

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: In your project configuration, the option is on the Code Prover Verification node. See "Dependencies" on page 1-171 for other options that you must also enable.

Command line: Use the option `-class-analyzer`. See "Command-Line Information" on page 1-171.

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 speci ed 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.
- 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: polyspace-code-prover-nodesktop -sources *file_name* -main-generator -class-analyzer custom=myClass1,myClass2

See Also

Verify module or library (-main-generator) | Functions to call within the specified classes (-class-analyzer-calls) | Analyze class contents only (-class-only) | Skip member initialization check (-no-constructors-init-check)

Topics

“Specify Polyspace Analysis Options” (Polyspace Code Prover)
“Verify C++ Classes” (Polyspace Code Prover)

Functions to call within the speci ed classes (-class-analyzer-calls)

Specify class methods that you want to verify

Description

This option a ects a Code Prover analysis only.

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: In your project configuration, the option is on the Code Prover Veri cation node. See “Dependencies” on page 1-175 for other options that you must also enable.

Command line: Use the option `-class-analyzer-calls`. See “Command-Line Information” on page 1-175.

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`.
- `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: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public`

See Also

`Verify module or library (-main-generator)` | `Class (-class-analyzer)` | `Analyze class contents only (-class-only)` | `Skip member initialization check (-no-constructors-init-check)`

Topics

“Specify Polyspace Analysis Options” (Polyspace Code Prover)
“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.

Specify that Polyspace must verify only methods of classes that you specify using the option `Class (-class-analyzer)`.

Set Option

User interface: In your project configuration, the option is on the Code Prover Verification node. See "Dependencies" on page 1-177 for other options that you must also enable.

Command line: Use the option `-class-only`. See "Command-Line Information" on page 1-177.

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`.
- `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

See Also

`Verify module or library (-main-generator)` | `Class (-class-analyzer)` | `Functions to call within the specified classes (-class-analyzer-`

`calls)|Skip member initialization check (-no-constructors-init-check)`

Topics

“Specify Polyspace Analysis Options” (Polyspace Code Prover)
“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.

Specify functions that you want the generated main to call ahead of other functions.

Set Option

User interface: In your project configuration, the option is on the Code Prover Verification node. See "Dependencies" on page 1-180 for other options that you must also enable.

Command line: Use the option -functions-called-before-main. See "Command-Line Information" on page 1-181.

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: polyspace-code-prover-nodesktop -sources *file_name* -main-generator -functions-called-before-main myfunc
Example: polyspace-code-prover-nodesktop -sources *file_name* -main-generator -functions-called-before-main myClass::init(int)

See Also

Verify module or library (-main-generator)|Variables to initialize (-main-generator-writes-variables)|Functions to call (-main-generator-calls)|Class (-class-analyzer)|Functions to call within the specified classes (-class-analyzer-calls)|Analyze class contents only (-class-only)

Topics

“Verify C Application Without main Function” (Polyspace Code Prover)

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.

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: 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 1-183.

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.

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

`Verify module or library (-main-generator)`

Topics

“Verify C Application Without main Function” (Polyspace Code Prover)

Main entry point (-main)

Specify a Microsoft Visual C++ extensions of main

Description

This option affects a Code Prover analysis only.

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: 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 -main. See "Command-Line Information" on page 1-185.

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:` `polyspace-code-prover-nodesktop -sources` *`file_name`* `-compiler visual14.0 -main _tmain`

See Also

`Verify module or library (-main-generator)`

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.

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: 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 `-main-generator-calls`. See "Command-Line Information" on page 1-188.

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.

Command-Line Information

Parameter: `-main-generator-calls`
Value: `none` | `unused` | `all` | `custom=`*`function1`*`[,`*`function2`*`[,...]]`
Default: `unused`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator -main-generator-calls all`

See Also

`Verify module or library (-main-generator)` | `Variables to initialize (-main-generator-writes-variables)` | `Initialization functions (-functions-called-before-main)` | `Class (-class-analyzer)` | `Functions to call within the specified classes (-class-analyzer-calls)` | `Analyze class contents only (-class-only)`

Topics

“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.

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: 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 -main-generator-writes-variables. 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 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.

Command-Line Information

Parameter: `-main-generator-writes-variables`
Value: `uninit | none | public | all | custom=`*`variable1`*`[,`*`variable2`*`[,...]]`
Default: (C) `public` | (C++) `uninit`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator -main-generator-writes-variables all`

See Also

`Verify module or library (-main-generator)` | `Initialization functions (-functions-called-before-main)` | `Functions to call (-main-generator-`

calls)|Class (-class-analyzer)|Functions to call within the specified classes (-class-analyzer-calls)|Analyze class contents only (-class-only)

Topics

“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: In your project configuration, the option is on the Code Prover Verification node. See "Dependencies" on page 1-193 for other options that you must also enable.

Command line: Use the option `-no-constructors-init-check`. See "Command-Line Information" on page 1-193.

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`.
- `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

See Also

`Verify module or library (-main-generator)` | `Class (-class-analyzer)`

Topics

"Specify Polyspace Analysis Options" (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.

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, 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.

Set Option

User interface: In your project configuration, the option is on the Code Prover Verification node. See “Dependencies” on page 1-195 for other options that you must also enable.

Command line: Use the option `-unit-by-unit`. See “Command-Line Information” on page 1-195.

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

- 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 the files:

  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`.

Command-Line Information

`Parameter:` `-unit-by-unit`
`Default:` Off
`Example:` `polyspace-code-prover-nodesktop -sources` *`file_name`* `-unit-by-unit`

See Also

Common source files (-unit-by-unit-common-source)

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.

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: 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 -unit-by-unit-common-source. See “Command-Line Information” on page 1-198.

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: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-unit-by-unit -unit-by-unit-common-source definitions.c`

See Also

`Verify files independently (-unit-by-unit)`

Verify model generated code (-main-generator)

Specify that a main function must be generated if it is not present in source files

Description

This option is available only for model-generated code.

Specify that Polyspace must generate a main function if it does not find one in the source files.

Set Option

User interface: 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 1-200.

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: `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-main-generator ...`

See Also

`Parameters (-variables-written-before-loop)` | `Inputs (-variables-written-in-loop)` | `Initialization functions (-functions-called-before-loop)` | `Step functions (-functions-called-in-loop)` | `Termination functions (-functions-called-after-loop)`

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

This option is available only for model- generated code.

Specify functions that the generated main must call before the cyclic code begins.

Set Option

User interface: In your project configuration, the option is available on the Main Generator node.

Command line: Use the option -functions-called-before-loop. See "Command-Line Information" on page 1-201.

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().

Command-Line Information

Parameter: -functions-called-before-loop
No Default

Value: *function1*[,*function2*[,...]]
Example: `polyspace-bug-finder-nodesktop -sources file_name -main-generator -functions-called-before-loop myfunc`

See Also

`Parameters (-variables-written-before-loop)` | `Inputs (-variables-written-in-loop)` | `Step functions (-functions-called-in-loop)` | `Termination functions (-functions-called-after-loop)`

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

This option is available only for model-generated code.

Specify functions that the generated `main` must call in each cycle of the cyclic code.

Set Option

User interface: In your project configuration, the option is available on the Main Generator node.

Command line: Use the option `-functions-called-in-loop`. See “Command-Line Information” on page 1-204.

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` or `Termination functions`, 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: `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-main-generator -functions-called-in-loop all`

See Also

`Parameters (-variables-written-before-loop)` | `Inputs (-variables-written-in-loop)` | `Initialization functions (-functions-called-before-loop)` | `Termination functions (-functions-called-after-loop)`

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

This option is available only for model-generated code.

Specify functions that the generated main must call after the cyclic code ends.

Set Option

User interface: In your project configuration, the option is available on the Main Generator node.

Command line: Use the option -functions-called-after-loop. See "Command-Line Information" on page 1-206.

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, you cannot specify it for Termination functions.

Command-Line Information

Parameter: `-functions-called-after-loop`
No Default
Value: *`function1`*`[,`*`function2`*`[,...]]`
Example: `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-main-generator -functions-called-after-loop myfunc`

See Also

`Parameters (-variables-written-before-loop)` | `Inputs (-variables-written-in-loop)` | `Initialization functions (-functions-called-before-loop)` | `Step functions (-functions-called-in-loop)`

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

This option is available only for model-generated code.

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: In your project configuration, the option is available on the Main Generator node.

Command line: Use the option `-variables-written-before-loop`. See "Command-Line Information" on page 1-208.

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: `public`
Example: `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-main-generator -variables-written-before-loop all`

See Also

`Inputs (-variables-written-in-loop)` | `Initialization functions (-functions-called-before-loop)` | `Step functions (-functions-called-in-loop)` | `Termination functions (-functions-called-after-loop)`

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

This option is available only for model-generated code.

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: In your project configuration, the option is available on the Main Generator node.

Command line: Use the option `-variables-written-in-loop`. See “Command-Line Information” on page 1-210.

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: `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-main-generator -variables-written-in-loop all`

See Also

`Parameters (-variables-written-before-loop)` | `Initialization functions (-functions-called-before-loop)` | `Step functions (-functions-called-in-loop)` | `Termination functions (-functions-called-after-loop)`

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: 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 1-212.

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 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: `polyspace-bug-finder-nodesktop -sources file_name -main-generator ...`

See Also

`Parameters (-variables-written-before-loop)` | `Inputs (-variables-written-in-loop)` | `Initialization functions (-functions-called-before-loop)` | `Step functions (-functions-called-in-loop)` | `Termination functions (-functions-called-after-loop)`

Topics

"Specify Polyspace Analysis Options"

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: 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 1-217.

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 == }`	`void main(){ struct myStruct structVal; structVal.field1 = 1; 1);assert(structVal.field1 ==1); }`
Red	`void main(){ struct myStruct structVal; structVal.field1 = 1; assert(structVal.field1 != }`	`void main(){ struct myStruct structVal; structVal.field1 = 1; 1);assert(structVal.field1 !=1); }`
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: `polyspace-code-prover-nodesktop -sources file_name -consider-volatile-qualifier-on-fields`

See Also

Topics

"Specify External Constraints" (Polyspace Code Prover)

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: 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 1-221.

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 *`matlabroot`*`\polyspace\verifier\code_prover`. Here, *`matlabroot`* is the MATLAB installation folder, for instance, `C:\Program Files\MATLAB\R2017b`.

  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 Over ow 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 Over ow check is green.	Besides to-nearest mode, the Over ow 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 Over ow 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 Over ow 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);`

Rounding mode: to-nearest	Rounding mode: all
	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: `polyspace-code-prover-nodesktop -sources file_name -float-rounding-mode all`

See Also

`Overflow`

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: 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 1-223.

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

See Also

`Respect types in global variables (-respect-types-in-globals)` | `Non-initialized local variable`

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: 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 1-225.

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 o	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

See Also

`Respect types in fields (-respect-types-in-fields)` | `Non-initialized local variable`

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.

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: 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 1-229.

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:` `polyspace-code-prover-nodesktop -sources` *`file_name`* `-stubbed-pointers-are-unsafe`

See Also

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

Topics

“Specify External Constraints” (Polyspace Code Prover)
“External Constraints for Polyspace Analysis” (Polyspace Code Prover)

`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: 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 1-231.

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

See Also

Invalid shift operations

Consider non nite oats (-allow-non-finite-floats)

Enable a verification mode that incorporates infinities and NaNs

Description

This option a ects a Code Prover analysis only. If you run a Bug Finder analysis, disable this option or the checkers for over ow and division by zero. See Numerical Defects .

Enable a verification mode that incorporates infinities and NaNs for floating point operations.

Set Option

User interface: 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 1-235.

Why Use This Option

By default, the verification does not incorporate infinities and NaNs. For instance, the verification terminates the execution thread where a division by zero occurs and does not consider that the result could be infinity.

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`.

Settings

☑ On

The verification allows infinities and NaNs. For instance, in this mode:

- The verification 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.
- The verification 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 verification does not allow infinities and NaNs. For instance, in this mode:

- The verification 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 verification assumes that floating-point variables with unknown values are full-range but finite.

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.

- If you select this option, the number and type of 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

Polyspace 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, Polyspace 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 verification 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 verification 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 if you incorporate non-finites in your verification.

Command-Line Information

Parameter: -allow-non-finite-floats
Default: Off

See Also

Infinities (-check-infinite) | NaNs (-check-nan) | Division by zero | Overflow | Invalid shift operations | Invalid use of standard library routine

Topics

“Specify Polyspace Analysis Options” (Polyspace Code Prover)

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: In your project configuration, the option is on the Check Behavior node. See "Dependencies" on page 1-238 for other options you must also enable.

Command line: Use the option `-check-infinite`. See "Command-Line Information" on page 1-238.

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: `polyspace-code-prover-nodesktop -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`

Introduced in R2016a

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: In your project configuration, the option is on the Check Behavior node. See “Dependencies” on page 1-241 for other options you must also enable.

Command line: Use the option -check-nan. See “Command-Line Information” on page 1-241.

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 oats` check for NaN.

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
    return y;
}
```

Even though the `Invalid operation on oats` 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 oats` 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: polyspace-code-prover-nodesktop -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

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: In your project configuration, the option is on the Check Behavior node. See “Dependency” on page 1-243 for other options you must also enable.

Command line: Use the option `-allow-ptr-arith-on-struct`. See “Command-Line Information” on page 1-244.

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: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-allow-ptr-arith-on-struct`

See Also

`Allow incomplete or partial allocation of structures (-size-in-bytes)` | `Illegally dereferenced pointer`

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: 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 1-247.

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:

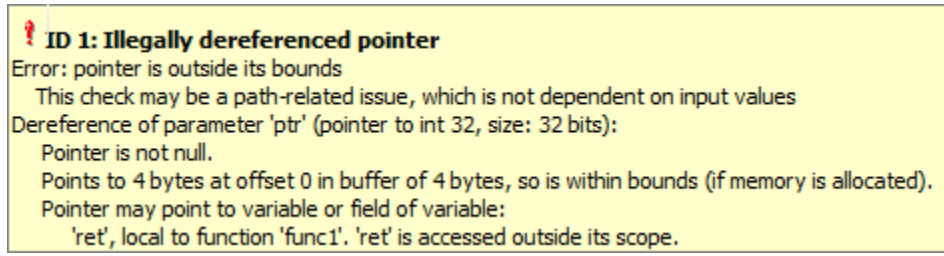
- Red, if all the variables that the pointer points to are accessed outside their scope.

  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 requirements:

- The pointer is not NULL.
- The pointer points within the memory buffer.

Command-Line Information

Parameter: -detect-pointer-escape
Default: Off

See Also

Illegally dereferenced pointer

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: 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 1-250.

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 (Polyspace Code Prover), 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 Over ow 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 Over ow 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: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-disable-initialization-checks`

See Also

Ignore over owing computations on constants (-ignore-constant-overflows)

Allow overflow in computations involving constants

Description

This option a ects a Code Prover analysis only.

Specify that the verification must allow overflow in computations involving constants.

Set Option

User interface: In your project configuration, the option is on the Check Behavior node.

Command line: Use the option -ignore-constant-overflows. See "Command-Line Information" on page 1-252.

Why Use This Option

Overflows in computations with compile-time constants can stop the analysis. Use this option to ignore these overflows and continue the analysis.

For instance, `char x = 0xff;` causes an overflow according to the ANSI C standard. However, if you use this option, Polyspace considers that this statement is equivalent to `char x = -1;`.

Settings

☑ On

The verification allows overflows in computations involving constants.

☐ Off (default)

If an overflow occurs in computations involving constants, the verification can stop.

Tips

- This option applies to computations involving compile-time constants only. For instance, the statement `char x = (rand() ? 0xFF:0xFE);` causes an `Overflow` error irrespective of whether the option is used because the value of `x` is not known at compile-time.

Command-Line Information

Parameter: `-ignore-constant-overflows`
Default: Off

See Also

`Overflow`

Permissive function pointer calls (-permissive-function-pointer)

Allow type mismatch between function pointers and the functions they point to

Description

This option affects 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: 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 -permissive-function-pointer. See "Command-Line Information" on page 1-254.

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*)`.

☐ Off (default)

The verification must require that the argument and return types of a function pointer and the function it calls are identical.

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.

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: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-permissive-function-pointer`

See Also

Over ow computation mode (-scalar-overflows-behavior)

Specify whether result of overflow must be wrapped around or truncated

Description

This option a ects a Code Prover analysis only.

Specify whether Polyspace must wrap the result of an integer overflow or restrict it to its extremum value.

Set Option

User interface: In your project configuration, the option is on the Check Behavior node.

Command line: Use the option -scalar-overflows-behavior. See “Command-Line Information” on page 1-256.

Why Use This Option

Use this option to specify the assumptions to make following an integer overflow.

Settings

Default: truncate-on-error

truncate-on-error

If the Over ow 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. However, Polyspace considers that:

- After a positive `Over ow`, 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 `Over ow`, the result of the operation has a lower bound. This lower bound is the minimum value allowed by the type of the result.

`wrap-around`

Polyspace analyzes the remaining code in the current scope even after a red integer `Over ow`. However, Polyspace wraps the result of the overflow. For instance, if you choose this option:

- In the following code, after the red `Over ow`, Polyspace considers that `i` has a value -2^{31}.

```
#include<stdio.h>

void main() {
 int i=1;
 i = i << 30;
 i = i *2;
 printf("%d",i);
}
```

- In the following code, before the orange `Over ow`, `i` has values in the range $[1..2^{31}-1]$. But, after the orange `Over ow`, Polyspace considers that `i` has even values in the range $[-2^{31}..2] \text{ or } [2..2^{31}-2]$.

```
#include<stdio.h>
int getVal();

void main() {
 int i=getVal();
 if(i>0) {
  i = i*2;
  printf("%d",i);
 }
}
```

Command-Line Information

Parameter: `-scalar-overflows-behavior`
Value: `wrap-around | truncate-on-error`
Default: `truncate-on-error`

Example: `polyspace-code-prover-nodesktop -sources file_name -scalar-overflows-behavior wrap-around`

See Also

`Detect overflows (-scalar-overflows-checks)` | `Overflow`

Detect overflows (-scalar-overflows-checks)

Specify whether to check for integer overflows on signed and unsigned variables

Description

This option affects a Code Prover analysis only.

Specify whether to check for integer overflows on signed and unsigned variables.

Set Option

User interface: In your project configuration, the option is on the Check Behavior node.

Command line: Use the option `-scalar-overflows-checks`. See "Command-Line Information" on page 1-259.

Why Use This Option

Use this option to specify the kinds of integer overflows that the verification must detect.

Settings

Default: `signed`

`signed`

The verification checks for overflows in computations involving signed integers alone. This behavior conforms to the ANSI C (ISO C++) standard.

`signed-and-unsigned`

The verification checks for overflows in all integer computations. This behavior is stricter than the ANSI C (ISO C++) standard.

`none`

The verification does not check for integer overflows. If a computed value exceeds the range of its type, the value is wrapped. For instance, in the following code, `x` is wrapped to 0 after the sum.

```
unsigned char x;
x = 255;
x = x+1;
```

Tips

- Following an overflow, unless you select `none`, Polyspace can either wrap the result or restrict it to its extremum value. Use `Overflow computation mode` to specify how the verification handles results of an overflow.
- Use the option `signed-and-unsigned` if you are computing the size of a buffer from `unsigned` integers. Using this option helps you detect an overflow at the buffer computation stage. Otherwise, you might see an error later due to insufficient buffer.
- If you use the option `signed-and-unsigned`, Polyspace does not produce an overflow error on bitwise NOT operations if you cast the result of the operation back to the operand type. For instance, Polyspace does not produce an overflow error on `(uint8_t)(~var)` where `var` is of type `uint8_t`.

Command-Line Information

Parameter: `-scalar-overflows-checks`
Value: `signed` | `signed-and-unsigned` | `none`
Default: `signed`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-scalar-overflows-checks signed`

See Also

`Overflow computation mode (-scalar-overflows-behavior)` | `Overflow`

Topics

“Detect Overflows in Buffer Size Computation” (Polyspace Code Prover)

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 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: 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 1-262.

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: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-size-in-bytes`

See Also

`Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)` | `Illegally dereferenced pointer`

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: 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 1-266.

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: `polyspace-code-prover-nodesktop -sources file_name -check-subnormal forbid`

See Also

Polyspace Results
`Subnormal float`

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: 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 1-269.

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: `polyspace-code-prover-nodesktop -sources file_name -uncalled-function-checks all`

See Also

`Function not called` | `Function not reachable`

Topics

“Specify Polyspace Analysis Options” (Polyspace Code Prover)
“Review and Fix Function Not Called Checks” (Polyspace Code Prover)
“Review and Fix Function Not Reachable Checks” (Polyspace Code Prover)

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: 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, you can identify 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 .

Command-Line Information

Parameter: `-context-sensitivity`
Value: *`function1`*`[,`*`function2`*`,...]`
Default: `none`
Example: `polyspace-code-prover-nodesktop -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” (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: 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 1-273.

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

See Also

Topics

"Improve Verification Precision" (Polyspace Code Prover)

Precision level (-O)

Specify a precision level for the verification

Description

This option a ects a Code Prover analysis only.

Specify the precision level that the verification must use.

Set Option

User interface: In your project configuration, the option is available on the Precision node.

Command line: Use the option `-O#`, for instance, `-O0` or `-O1`. See “Command-Line Information” on page 1-275.

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 complex polyhedron model of domain values.

2

This option corresponds to more complex algorithms closely modelling domain values. The algorithms combine both complex polyhedrons and integer lattices.

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.

Command-Line Information

Parameter: `-O0` | `-O1` | `-O2` | `-O3`
Default: `-O2`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-O1`

See Also

Topics

“Specify Polyspace Analysis Options” (Polyspace Code Prover)
“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: In your project configuration, the option is available on the Precision node. See “Dependency” on page 1-277 for other options you must also enable.

Command line: Use the option `-modules-precision`. See “Command-Line Information” on page 1-277.

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: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-O1 -modules-precision My_File:02`

See Also

`Precision level (-O)`

Topics

"Specify Polyspace Analysis Options" (Polyspace Code Prover)
"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: In your project configuration, the option is available on the Precision node.

Command line: Use the option -to. See "Command-Line Information" on page 1-281.

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 exceptions (Polyspace Code Prover) only. If you check for coding rules alone, you can lower the verification level.
- 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 completes coding rules checking at the end of the compilation phase.

`Software Safety Analysis level 0`

The verification process runs once on your source code.

`Software Safety Analysis level 1`

The verification process runs twice on your source code.

`Software Safety Analysis level 2`

The verification process runs three time on your source code. Use this option for most accurate results in reasonable time.

`Software Safety Analysis level 3`

The verification process runs four times on your source code.

`Software Safety Analysis level 4`

The verification process runs five times on your source code.

`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.

  ### Difference between Level 0 and 1

  The following example illustrates the difference between `Software Safety Analysis level 0` and `Software Safety Analysis level 1`:

Software Safety Analysis Level 0	Software Safety Analysis Level 1

`#include <stdlib.h>`
 	
`void ratio (float x, float *y)`
`{`
`  *y=(abs(x-*y))/(x+*y);`
`}`
 	
`void level1 (float x,`
`  float y, float *t)`
`{ float v;`
`  v = y;`
`  ratio (x, &y);`
`  *t = 1.0/(v - 2.0 * x);`
`}`
 	
`float level2(float v)`
`{`
`  float t;`
`  t = v;`
`  level1(0.0, 1.0, &t);`
`  return t;`
`}`
 	
`void main(void)`
`{`
`  float r,d;`
`  d= level2(1.0);`
`  r = 1.0 / (2.0 - d);`
`}`
	`#include <stdlib.h>`
	
	`void ratio (float x, float *y)`
	`{`
	`  *y=(abs(x-*y))/(x+*y);`
	`}`
	
	`void level1 (float x,`
	`  float y, float *t)`
	`{ float v;`
	`  v = y;`
	`  ratio (x, &y);`
	`  *t = 1.0/(v - 2.0 * x);`
	`}`
	
	`float level2(float v)`
	`{`
	`  float t;`
	`  t = v;`
	`  level1(0.0, 1.0, &t);`
	`  return t;`
	`}`
	
	`void main(void)`
	`{`
	`  float r,d;`
	`  d= level2(1.0);`
	`  r = 1.0 / (2.0 - d);`
	`}`

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 you want to check for coding rules only, you can run Polyspace on your source code up to the `Source Compliance Checking` phase.

  With the exception of certain rules (Polyspace Code Prover) Polyspace checks for coding rule violations during the compilation phase.
- 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.

Command-Line Information

Parameter: `-to`
Value: `compile` | `pass0` | `pass1` | `pass2` | `pass3` | `pass4` | `other`
Default: `pass2`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-to pass2`

See Also

Topics

“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: In your project configuration, the option is available on the Precision node.

Command line: Use the option -timeout. See “Command-Line Information” on page 1-283.

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: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-timeout 5.75`

See Also

Topics

“Specify Polyspace Analysis Options” (Polyspace Code Prover)
“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: In your project configuration, the option is available on the Scaling node.

Command line: Use the option -inline. See "Command-Line Information" on page 1-286.

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 worst color is shown for the check. Therefore, the check is red.

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: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-inline func1,func2`

See Also

Depth of veri cation inside structures (-k-limiting)

Limit the depth of analysis for nested structures

Description

This option a ects a Code Prover analysis only.

Specify a limit to the depth of analysis for nested structures.

Set Option

User interface: 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 1-288.

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`*
Default: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-k-limiting 3`

See Also

Generate report

Specify whether to generate a report after the analysis

Description

Specify whether to generate a report after the analysis.

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: In your project configuration, the option is available on the `Reporting` node.

Command line: See “Command-Line Information” on page 1-290.

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.

☐ Off (default)

Polyspace does not generate an analysis report. You can still view your results in the Polyspace interface.

Tips

- To generate a report after an analysis is complete, select Reporting > Run Report. Alternatively, at the command line, use the command `polyspace-report-generator` with the options `-template` and `-format`.

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)`

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 *`matlabroot`*`\toolbox\polyspace\psrptgen\templates\`. Here, *`matlabroot`* is the MATLAB installation folder.

Set Option

User interface: 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 1-298 for other options you must also enable.

Command line: Use the option `-report-template`. See "Command-Line Information" on page 1-298.

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. 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.

`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. 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.

`CodingRules`

For C code, the report lists information about compliance with:

- MISRA C rules
- MISRA AC AGC rules
- Custom coding rules

For C++ code, the report lists information about compliance with:

- MISRA C++ rules
- JSF C++ rules
- Custom coding rules

This report also contains the Polyspace configuration settings for the analysis. An additional section states the rules along with the information whether they were enabled or disabled.

`SecurityCERT`

The report contains the same information as the `BugFinder` report. However, in the Defects chapter, an additional column lists the CERT® C rules mapped to each defect. The Con guration Settings appendix also includes a Security Standard to Polyspace Result Map.

`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 Con guration Settings appendix also includes a Security Standard to Polyspace Result Map.

`SecurityISO_17961`

The report contains the same information as the `BugFinder` report. However, in the Defects chapter, an additional column lists the ISO/IEC TS 17961 rules mapped to each defect. The Con guration 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.

CodingRules

For C code, the report lists information about compliance with:

- MISRA C rules
- MISRA AC AGC rules
- Custom coding rules

For C++ code, the report lists information about compliance with:

- MISRA C++ rules
- JSF C++ rules
- Custom coding rules

This report also contains the Polyspace configuration settings and modifiable assumptions used in the analysis. An additional section states the rules along with the information whether they were enabled or disabled.

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 downloaded from the Polyspace Metrics web dashboard.

`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 downloaded from the Polyspace Metrics web dashboard.

Dependencies

This option is available only if you select the `Generate report` check box.

Tips

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: `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-report-template` *`matlabroot`*`\toolbox\polyspace\psrptgen\templates\bug_finder\BugFinder.rpt`

See Also

`Generate report` | `Output format (-report-output-format)`

Topics

“Generate Reports”

Output format (-report-output-format)

Specify output format of generated report

Description

Specify output format of analysis report.

Set Option

User interface: In your project configuration, the option is on the Reporting node. See “Dependencies” on page 1-301 for other options you must also enable.

Command line: Use the option -report-output-format. See “Command-Line Information” on page 1-301.

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

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

This option is enabled only if you select the `Generate report` box.

Command-Line Information

Parameter: `-report-output-format`
Value: `html | pdf | word`
Default: `word`
Example: `polyspace-bug-finder-nodesktop -sources file_name -report-output-format pdf`

See Also

`Generate report` | `Bug Finder and Code Prover report (-report-template)`

Topics

"Specify Polyspace Analysis Options"
"Generate Reports"

Run Bug Finder or Code Prover analysis on a remote cluster (-batch)

Enable batch remote analysis

Description

Enable batch remote analysis.

For batch remote analysis, you need:

- Polyspace and MATLAB Distributed Computing Server® on the cluster
- MATLAB, Polyspace and Parallel Computing Toolbox® on your local computer.

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 1-303.

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.
- On the DOS or UNIX® command line, use the `polyspace-jobs-manager` command. For more information, see "Run Polyspace Analysis on Remote Clusters 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.

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: `polyspace-code-prover-nodesktop -batch -scheduler NodeHost`
`polyspace-code-prover-nodesktop -batch -scheduler MJSName@NodeHost`
Example: `polyspace-bug-finder-nodesktop -batch -scheduler NodeHost`
`polyspace-bug-finder-nodesktop -batch -scheduler MJSName@NodeHost`

See Also

`Upload results to Polyspace Metrics (-add-to-results-repository)|-scheduler`

Topics

“Specify Polyspace Analysis Options”
“Set Up Server for Metrics and Remote Analysis”
“Run Polyspace Analysis on Remote Clusters”

Upload results to Polyspace Metrics (-add-to-results-repository)

Upload analysis results for viewing on Polyspace Metrics web dashboard

Description

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 1-306 for other options that you must also enable.

Command line: Use the option -add-to-results-repository. See "Command-Line Information" on page 1-306.

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".

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: `polyspace-code-prover-nodesktop -batch -scheduler NodeHost -add-to-results-repository -password` *`passwordName`*
Example: `polyspace-bug-finder-nodesktop -batch -scheduler NodeHost -add-to-results-repository -password` *`passwordName`*

The password is optional.

See Also

`Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`

Topics

“Upload Results to Polyspace Metrics”
“View Projects in 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 a Bug Finder fast-analysis, Polyspace analyzes each file for a subset of defects and coding rules. These defects and rules are coding errors that can be found within a single compilation unit, such as a single function or file. The software does not perform interprocedural or cross-functional analysis.

Set Option

User interface: 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 1-309.

Why Use This Option

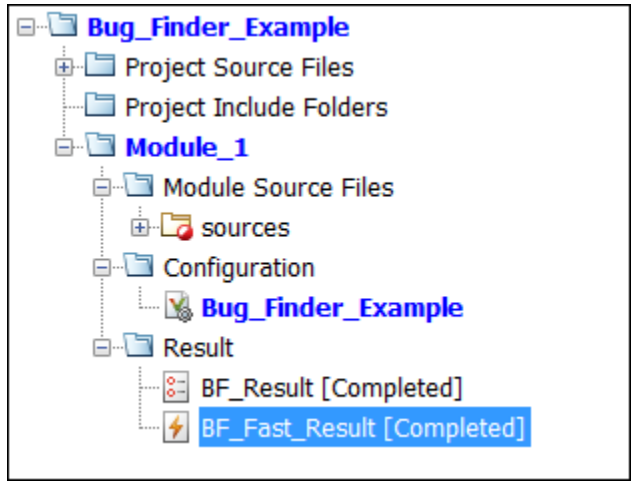
If you use this option, you have to wait less for analysis results from your second analysis onwards. During development, you can frequently run analysis in fast mode and quickly check for new defects or coding rule violations.

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
Source file modified	Modified source file
Header file modified	Source files that include the modified header file (directly or indirectly)
Analysis options added or removed	All files
Previous fast-analysis results not found	All files

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

In fast analysis mode:

- 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.
- 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".
- You cannot specify constraints using the option Constraint setup (-data-range-specifications).
- You cannot run analysis on a remote cluster.

Command-Line Information

Parameter: -fast-analysis
Default: Off
Example: polyspace-bug-finder-nodesktop -sources *filename* -fast-analysis

See Also

"Defects"

Topics

"Results Found by Fast Analysis"

Command/script to apply after the end of the code veri cation (-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: 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 1-312.

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.

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 *`matlabroot`*`\sys\perl\win32\bin\perl.exe` *`<absolute_path>`*`\send_email.pl`. Here, *`matlabroot`* is the location of the current MATLAB installation, such as `C:\Program Files\MATLAB\R2015b\`, and *`<absolute_path>`* is the location of the Perl script.

Tips

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.

Command-Line Information

Parameter: `-post-analysis-command`
Value: Path to executable file or command in quotes
No Default
Example in Linux: `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-post-analysis-command \`pwd\`/send_email.pl`
Example in Windows: `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-post-analysis-command "C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\send_email"`

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.

Specify that the Automatic Orange Tester must be executed at the end of the verification.

Set Option

User interface: In your project configuration, the option is on the Advanced Settings node. See "Dependency" on page 1-315 for other options you must also enable.

Command line: Use the option `-automatic-orange-tester`. See "Command-Line Information" on page 1-315.

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 nite oats.

Command-Line Information

Parameter: -automatic-orange-tester
Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -lang c -automatic-orange-tester`

See Also

`Number of automatic tests (-automatic-orange-tester-tests-number)` | `Maximum loop iterations (-automatic-orange-tester-loop-max-iteration)` | `Maximum test time (-automatic-orange-tester-timeout)`

Topics

“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.

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: In your project configuration, the option is on the Advanced Settings node. See “Dependencies” on page 1-317 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.

- Turn on `Automatic Orange Tester (-automatic-orange-tester)`.

Command-Line Information

Parameter: `-automatic-orange-tester-loop-max-iteration`
Value: *`positive integer`*
Default: 1000
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-lang c -automatic-orange-tester -automatic-orange-tester-loop-max-iteration 500`

See Also

`Automatic Orange Tester (-automatic-orange-tester)` | `Number of automatic tests (-automatic-orange-tester-tests-number)` | `Maximum test time (-automatic-orange-tester-timeout)`

Topics

“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.

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: 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 1-320.

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.
- Turn on Automatic Orange Tester (-automatic-orange-tester).

Command-Line Information

Parameter: `-automatic-orange-tester-tests-number`
Value: *`positive integer`*
Default: 500
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-lang c -automatic-orange-tester -automatic-orange-tester-tests-number 500`

See Also

`Automatic Orange Tester (-automatic-orange-tester)` | `Maximum loop iterations (-automatic-orange-tester-loop-max-iteration)` | `Maximum test time (-automatic-orange-tester-timeout)`

Topics

"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.

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: In your project configuration, the option is on the Advanced Settings node. See “Dependencies” on page 1-321 for other options you must also enable.

Command line: Use the option -automatic-orange-tester-timeout. See “Command-Line Information” on page 1-322.

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.
- Turn on Automatic Orange Tester (-automatic-orange-tester).

Command-Line Information

Parameter: `-automatic-orange-tester-timeout`
Value: *`time`*
Default: 5
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-lang c -automatic-orange-tester -automatic-orange-tester-test-timeout 10`

See Also

`Automatic Orange Tester (-automatic-orange-tester)` | `Number of automatic tests (-automatic-orange-tester-tests-number)` | `Maximum loop iterations (-automatic-orange-tester-loop-max-iteration)`

Topics

"Test Orange Checks for Run-Time Errors" (Polyspace Code Prover)

Other

Specify additional flags for analysis

Description

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.

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: `polyspace-bug-finder-nodesktop -extra-flags -option-name -extra-flags option_param`

2

Polyspace Command-Line Options

-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.

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, 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:

```
polyspace-bug-finder-nodesktop -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

`polyspaceBugFinder`

Topics

“Run Polyspace Analysis from Command Line”

-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 Polyspace user interface, 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.

```
polyspace-bug-finder-nodesktop -author "John Smith"
```

See Also

`-date` | `-prog` | `polyspaceBugFinder`

Topics

“Run Polyspace Analysis from Command Line”

-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.

```
polyspace-bug-finder-nodesktop -date "15/03/2012"
```

See Also

`-author` | `-prog` | `polyspaceBugFinder` | `polyspaceCodeProver`

Topics

“Run Polyspace Analysis from Command Line”

-function-behavior-specifications

Map imprecisely analyzed function to standard function for precise analysis

Syntax

`-function-behavior-specifications` *`file_path`*

Description

`-function-behavior-specifications` *`file_path`* specifies the path to an XML file. You can use this XML file to map some of your functions to corresponding standard functions that Polyspace recognizes. If you run verification from the command line, *`file_path`* is the absolute path or path relative to the folder from which you run the command. If you run verification from the user interface, *`file_path`* is the absolute path.

If you are running an analysis from the user interface, on the Configuration pane, you can enter this option in the Other field. See `Other`.

Using Option for Precision Improvement

This section applies only to a Code Prover analysis.

Use this option to reduce the number of orange checks from imprecise analysis of your function. 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].

A sample file `function-behavior-specifications-sample.xml` shows the functions that you can map to. The file is in *`matlabroot`*`\polyspace\verifier\cxx\` where *`matlabroot`* is the MATLAB installation folder. 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. See “Stub lookup tables” (Polyspace Code Prover). 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.

Using Option for Concurrency Detection

This section applies both to a Bug Finder and a Code Prover analysis.

Use this option 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 option.

If your thread-creation function, for instance, does not belong to one of the supported families, map your function to a supported concurrency primitive.

To find which multitasking primitives can be automatically detected, see “Auto-Detection of Thread Creation and Critical Section in Polyspace”.

Examples

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 `function-behavior-specifications-sample.xml` from *`matlabroot`*`\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.

```
polyspace-code-prover-nodesktop -function-behavior-specifications
  "C:\Polyspace_projects\Common\Config_files
   \function-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 `function-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 `function-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>
```

E ect 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 speci cation:

    ```
    <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 speci cation:

    ```
    <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 speci cation:

```
<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 speci cation:

```
<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;
```

Effect of Mapping on Concurrency Detection

In this example, the Polyspace support for automatic concurrency detection is extended by mapping unsupported functions to the supported `Pthreads` functions.

- Thread creation function: `createTask` → `pthread_create`
- Function that begins critical section: `takeLock` → `pthread_mutex_lock`
- Function that ends critical section: `releaseLock` → `pthread_mutex_unlock`

If you use the mapping, a Bug Finder analysis can determine the multitasking model used in your code and find possible race conditions.

- *Before mapping*:

  The analysis does not detect the data race on `var2`.

```
typedef void* (*FUNT) (void*);

extern int takeLock(int* t);
extern int releaseLock(int* t);
// First argument is the function, second the id
extern int createTask(FUNT,int*,int*,void*);

int t_id1,t_id2;
int lock;

int var1;
int var2;

void* task1(void* a) {
    takeLock(&lock);
    var1++;
    var2++;
    releaseLock(&lock);
    return 0;
}
```

```
void* task2(void* a) {
    takeLock(&lock);
    var1++;
    releaseLock(&lock);
    var2++;
    return 0;
}

void main() {
    createTask(task1,&t_id1,0,0);
    createTask(task2,&t_id2,0,0);
}
```

- Mapping speci cation:

  Based on the number and type of parameters of the function `createTask`, it is convenient to map `createTask` to the thread creation function `pthread_create`. The other available alternatives, `createThread` or `OSTaskCreate`, have different argument types.

  Even when mapping to `pthread_create`, argument remapping is required, because the arguments do not correspond exactly. The thread start routine is the third argument of `pthread_create` but the first argument of `createTask`.

  ```
  <function name="createTask" std="pthread_create" >
      <mapping std_arg="1" arg="2"></mapping>
      <mapping std_arg="3" arg="1"></mapping>
      <mapping std_arg="2" arg="3"></mapping>
      <mapping std_arg="4" arg="4"></mapping>
  </function>
  <function name="takeLock" std="pthread_mutex_lock" >
  </function>
  <function name="releaseLock" std="pthread_mutex_unlock" >
  </function>
  ```

  For the list of supported functions that you can map to, see the sample mapping file `function-behavior-specifications-sample.xml` in *`matlabroot`*`\polyspace\verifier\cxx\`. *`matlabroot`* is the MATLAB installation folder, such as `C:\Program Files\MATLAB\R2017b`. See also “Auto-Detection of Thread Creation and Critical Section in Polyspace”.

- After mapping:

  The analysis detects the data race on `var2`.

```
typedef void* (*FUNT) (void*);

extern int takeLock(int* t);
extern int releaseLock(int* t);
// First argument is the function, second the id
extern int createTask(FUNT,int*,int*,void*);

int t_id1,t_id2;
int lock;

int var1;
int var2;

void* task1(void* a) {
    takeLock(&lock);
    var1++;
    var2++;
    releaseLock(&lock);
    return 0;
}

void* task2(void* a) {
    takeLock(&lock);
    var1++;
    releaseLock(&lock);
    var2++;
    return 0;
}

void main() {
    createTask(task1,&t_id1,0,0);
    createTask(task2,&t_id2,0,0);
}
```

See Also

“Stub lookup tables” (Polyspace Code Prover)

Topics

“Reduce Orange Checks” (Polyspace Code Prover)

Introduced in R2016b

-generate-launching-script-for

Extract information from project file

Syntax

`-generate-launching-script-for` *`PRJFILE`*

Description

`-generate-launching-script-for` *`PRJFILE`* extracts information from the project file *`PRJFILE`* so that you can run an analysis from the command line. 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`.

```
polyspace -generate-launching-script-for myproject.psprj -bug-finder
```

See Also

Topics

“Create Command-Line Script from Project File”
“Run Polyspace Analysis from Command Line”

-h[elp]

Display list of possible options

Syntax

```
-h
-help
```

Description

`-h` and `-help` display the list of possible options in the shell window and the argument syntax.

Examples

Display the command-line help.

```
polyspace-bug-finder-nodesktop -h
polyspace-bug-finder-nodesktop -help
```

See Also

`polyspaceBugFinder`

Topics

“Run Polyspace Analysis from Command Line”

-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.

```
polyspace-bug-finder-nodesktop -I /com1/inc -I /com1/sys/inc
```

Because `./sources` is included automatically, this Polyspace command is equivalent to:

```
polyspace-bug-finder-nodesktop -I /com1/inc -I /com1/sys/inc
                                               -I ./sources
```

See Also

`polyspaceBugFinder`

Topics

“Run Polyspace Analysis from Command Line”

-import-comments

Import comments and justifications from previous analysis

Syntax

`-import-comments` *`resultsFolder`*

Description

`-import-comments` *`resultsFolder`* imports the comments and justifications from a previous analysis, as specified by the results folder. *`resultsFolder`* must be the same type of analysis you are running. For example, if you are running a Bug Finder analysis, you cannot import comments from a Code Prover verification.

If you are running an analysis from the user interface, on the Configuration pane, you can enter this option in the Other field. See `Other`.

Examples

Increment your project's version number (`-version`) and import comments from the previous results.

```
polyspace-bug-finder-nodesktop -version 1.3
       -import-comments C:\Results\myProj\1.2
```

See Also

`-version` | `polyspaceBugFinder`

-no-assumption-on-absolute-addresses

Remove assumption that absolute address usage is valid

Syntax

```
-no-assumption-on-absolute-addresses
```

Description

This option affects Code Prover analysis only.

`-no-assumption-on-absolute-addresses` removes the default assumption that absolute addresses used in your code are valid. If you use this option, the verification produces an orange `Absolute address usage` check when you assign an absolute address to a pointer. Otherwise, the check is green by default.

The type of the pointer to which you assign the address determines the initial value stored in the address. For instance, if you assign the address to an `int*` pointer, following this check, the verification assumes that the memory zone that the address points to is initialized with an `int` value. The value can be anything allowed for the data type `int`.

If you are running an analysis from the user interface, on the `Configuration` pane, you can enter this option in the `Other` field. See `Other`.

Examples

The use of option `-no-assumption-on-absolute-addresses` can increase the number of orange checks in your code. For instance, the following table shows an additional orange check with the option enabled.

Absolute Address Usage Green	Absolute Address Usage Orange

`void main() {`
`  int *p = (int *)0x32;`
`  int x;`
`  x=*p;`
`}`
 	
In this example, the software produces:
 	
• A green `Absolute address usage` check when the address `0x32` is assigned to a pointer `p`.
• A green `Illegally dereferenced pointer` check when the pointer `p` is read.
 	
`x` potentially has all values allowed for an `int` variable.
	`void main() {`
	`  int *p = (int *)0x32;`
	`  int x;`
	`  x=*p;`
	`}`
	
	In this example, the software produces:
	
	• An orange `Absolute address usage` check when the address `0x32` is assigned to a pointer `p`.
	• A green `Illegally dereferenced pointer` check when the pointer `p` is read.
	
	`x` potentially has all values allowed for an `int` variable.

For best use of the `Absolute address usage` check, leave this check green by default during initial stages of development. During integration stage, use the option `-no-assumption-on-absolute-addresses` and detect all uses of absolute memory addresses. Browse through them and make sure that the addresses are valid.

See Also

`polyspaceCodeProver`

Topics

“Run Polyspace Analysis from Command Line” (Polyspace Code Prover)

`Introduced in R2016a`

-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, the 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. When you start a verification, a message states the number of logical processors detected on your system.

If you are running an analysis from the user interface, on the Con guration pane, you can enter this option in the Other field. See Other.

Examples

Disable parallel processing during the analysis.

```
polyspace-bug-finder-nodesktop -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

`polyspaceBugFinder`

Topics

“Run Polyspace Analysis from Command Line”

-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 on page 1-116 and cyclic tasks on page 1-118 but can be interrupted by interrupts on page 1-121, preemptable or nonpreemptable.

To specify a function as a nonpreemptable cyclic task, you must first specify the following options:

- `Configure multitasking manually`
- `Tasks (-entry-points)` or `Cyclic tasks (-cyclic-tasks)`: Specify the function name.

The functions that you specify must have the prototype:

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

If you are running an analysis from the user interface, on the `Configuration` pane, you can enter this option in the `Other` field. See `Other`.

See Also

`-preemptable-interrupts` | `Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)` | `Tasks (-entry-points)` | `Critical section details (-`

`critical-section-begin -critical-section-end)|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”

Introduced in R2016b

-options- le

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:

```
#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
```

2 Run Polyspace using options in the file `listofoptions.txt`.

```
polyspace-bug-finder-nodesktop -options-file listofoptions.txt
```

See Also

`polyspaceBugFinder` | `polyspaceConfigure`

Topics

“Run Polyspace Analysis from Command Line”

-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 on page 1-121, preemptable or nonpreemptable.

To specify a function as a preemptable interrupt, you must first specify the following options:

- `Configure multitasking manually`
- `Interrupts (-interrupts)`: Specify the function name.

The functions that you specify must have the prototype:

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

If you are running an analysis from the user interface, on the `Configuration` pane, you can enter this option in the `Other` field. See `Other`.

See Also

`-non-preemptable-tasks` | `Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)` | `Tasks (-entry-points)` | `Critical section details (-critical-section-begin -critical-section-end)` | `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”

Introduced in R2016b

-prog

Specify name of project

Syntax

`-prog` *`projectName`*

Description

`-prog` *`projectName`* specifies the name of your Polyspace project. This name must use only letters, numbers, underscores (_), dashes (-), or periods (.).

Examples

Assign a session name to your Polyspace Project.

```
polyspace-bug-finder-nodesktop -prog MyApp
```

See Also

`-author` | `-date` | `polyspaceBugFinder`

Topics

“Run Polyspace Analysis from Command Line”

-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, on the `Configuration` pane, you can enter this option in the `Other` field. See `Other`.

Examples

Specify the name of the analysis report.

```
polyspace-bug-finder-nodesktop -report-template Developer
      -report-output-name Airbag_v3.doc
```

See Also

`Output format (-report-output-format)` | `Bug Finder and Code Prover report (-report-template)` | `polyspaceBugFinder`

Topics

"Generate Reports"

-results-dir

Specify the results folder

Syntax

```
-results-dir
```

Description

`-results-dir` 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, see “Run Polyspace Analysis on Desktop”.

Examples

Specify to store your results in the `RESULTS` folder.

```
polyspace-bug-finder-nodesktop -results-dir RESULTS ...
         export RESULTS=results_'date + %d%B_%HH%M_%A'
polyspace-bug-finder-nodesktop -results-dir 'pwd'/$RESULTS
```

See Also

`polyspaceBugFinder`

Topics

“Run Polyspace Analysis from Command Line”

-scheduler

Specify cluster or job scheduler

Syntax

`-scheduler` *`schedulingOption`*

Description

`-scheduler` *`schedulingOption`* specifies the head node of the cluster or MATLAB job scheduler on the node host. Use this command to manage the cluster, or to specify where to run batch analyses.

Examples

Run a batch analysis on a remote server.

```
polyspace-bug-finder-nodesktop -batch -scheduler NodeHost
polyspace-bug-finder-nodesktop -batch -scheduler 192.168.1.124:12400
polyspace-bug-finder-nodesktop -batch -scheduler MJSName@NodeHost

polyspace-job-manager listjobs -scheduler NodeHost
```

See Also

`polyspaceBugFinder` | `polyspaceJobsManager` | `polyspaceJobsManager`

Topics

“Run Polyspace Analysis on Remote Clusters Using Scripts”

-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`.

```
polyspace-bug-finder-nodesktop -sources mymain.c
     -sources funAlgebra.c -sources funGeometry.c
```

See Also

`polyspaceBugFinder`

Topics

“Run Polyspace Analysis from Command Line”

-sources-list- le

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 absolute path to a source file. For example:

```
C:\Sources\myfile.c
C:\Sources2\myfile2.c
```

Examples

Run analysis on files listed in `files.txt`.

```
polyspace-bug-finder-nodesktop -batch -scheduler NODEHOST
        -sources-list-file "C:\Analysis\files.txt"
polyspace-bug-finder-nodesktop -batch -scheduler NODEHOST
        -sources-list-file "/home/polyspace/files.txt"
```

See Also

`polyspaceBugFinder`

Topics

“Run Polyspace Analysis on Remote Clusters Using Scripts”

-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. If a remote analysis stops after compilation, for instance because of communication problems between the server and client computers, use this option.

When you perform a remote analysis:

1. On the local host computer, the Polyspace software performs code compilation and coding rule checking.
2. The Parallel Computing Toolbox™ software submits the analysis job to the MATLAB job scheduler (MJS) on the head node of the MATLAB Distributed Computing Server cluster.
3. The head node of the MATLAB Distributed Computing 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, 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.

```
polyspace-bug-finder-nodesktop -batch -scheduler localhost
      -submit-job-from-previous-compilation-results
```

See Also

`polyspaceBugFinder`

Topics

“Run Polyspace Analysis on Remote Clusters”
“Run Polyspace Analysis on Remote Clusters Using Scripts”

-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, 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-nodesktop -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, on the Configuration pane, you can enter this option in the Other field. See Other.

Examples

Store temporary files in the results folder.

```
polyspace-bug-finder-nodesktop -tmp-dir-in-results-dir
```

See Also

polyspaceBugFinder

Topics

“Run Polyspace Analysis from Command Line”

-v[ersion]

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.

```
polyspace-bug-finder-nodesktop -v
```

See Also

`polyspaceBugFinder`

Topics

“Run Polyspace Analysis from Command Line”

-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 comments 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 Con guration 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.

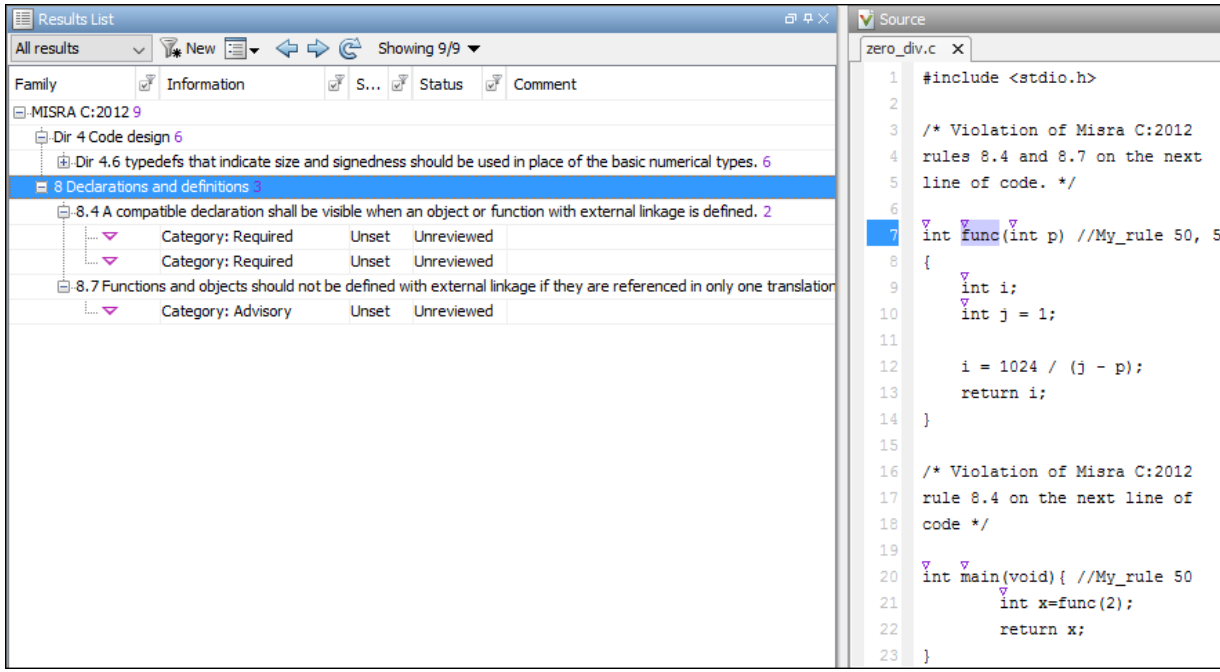
```
#include <stdio.h>

/* 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:

```
polyspace-bug-finder-nodesktop -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"
```

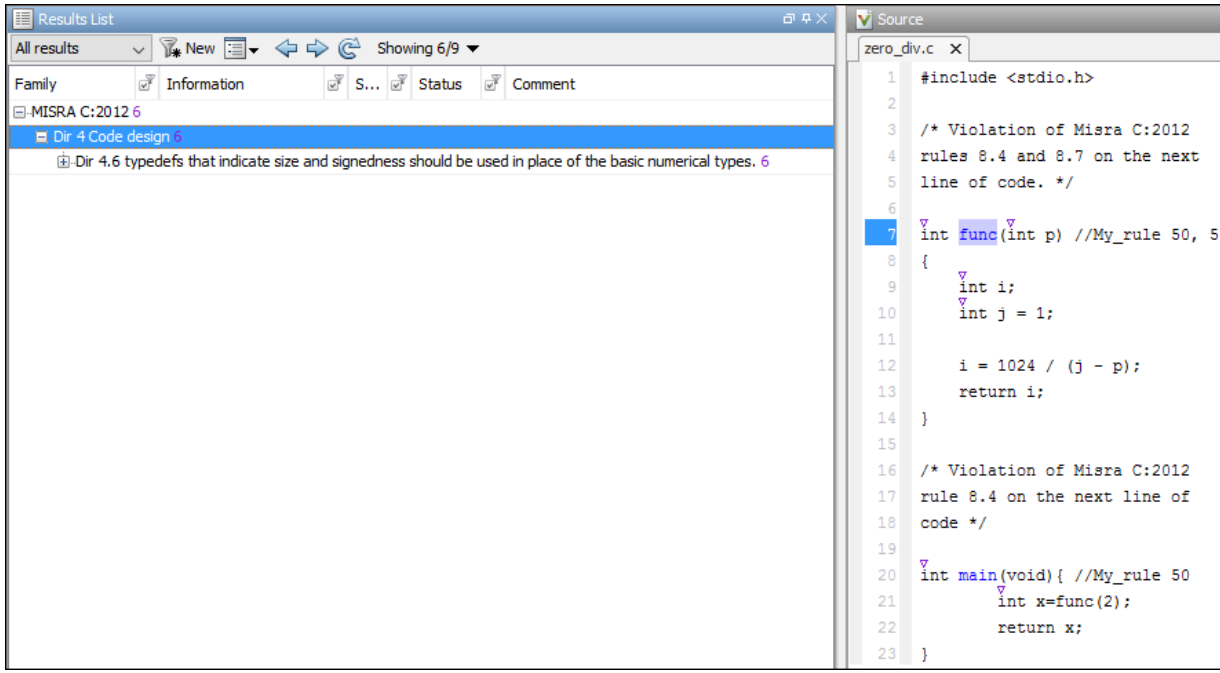
```
                Regex="My_rule\s(\w+(\s*,\s*\w+)*)"
                Rule_Identifier_Position="1"
                />

 </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:

   ```
   polyspace-bug-finder-nodesktop -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

“Define Custom Annotation Format”
“Annotation Description Full XML Template”

Introduced in R2017b

Defects

*this not returned in copy assignment operator

`operator=` method does not return a pointer to the current object

Description

`*this not returned from copy assignment operator` 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 3-3.
- 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.

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 Comments”

Introduced in R2015b

Abnormal termination of exit handler

Exit handler function interrupts the normal execution of a program

Description

`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.

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.

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
CERT C ID: ENV32-C

Introduced in R2016b

Absorption of oat operand

One addition or subtraction operand is absorbed by the other operand

Description

Absorption of oat 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.

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`.

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);
}
```

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: 682, 873
CERT C ID: FLP00-C

See Also

Polyspace Analysis Options
Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2016b

Accessing object with temporary lifetime

Read or write operations on the object are undefined behavior

Description

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.

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.

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
CERT C ID: EXP35-C

See Also

Topics

Large pass-by-value argument
Misuse of structure with flexible array member
Write without a further read

Introduced in R2018a

Alignment changed after memory reallocation

Memory reallocation changes the originally stricter alignment of an object

Description

`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.

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.

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
CERT C ID: MEM36-C

See Also

Introduced in R2017b

Alternating input and output from a stream without flush or positioning call

Undefined behavior for input or output stream operations

Description

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.

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.

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
CERT C ID: FIO39-C
ISO/IEC TS 17961 ID: `ioileave`

See Also

Introduced in R2017b

Arithmetic operation with NULL pointer

Arithmetic operation performed on NULL pointer

Description

Arithmetic operation with NULL pointer occurs when an arithmetic operation involves a pointer whose value is NULL.

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++.

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
CERT C ID: EXP34-C
ISO/IEC TS 17961 ID: nullref

See Also

Polyspace Analysis Options
Find defects (-checkers)

Polyspace Results
Null pointer

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Array access out of bounds

Array index outside bounds during array access

Description

`Array access out of bounds` occurs when an array index falls outside the range `[0...array_size-1]` during array access.

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`.

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
CERT C ID: ARR00-C, ARR30-C, ARR38-C, STR31-C, API02-C, MSC15-C
ISO/IEC TS 17961 ID: `invptr`

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Pointer access out of bounds`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Array access with tainted index

Array index from unsecure source possibly outside array bounds

Description

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.

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`.

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
CERT C ID: INT04-C, ARR30-C, API00-C, API02-C
ISO/IEC TS 17961 ID: `invptr`

See Also

`Loop bounded with tainted value` | `Pointer dereference with tainted offset` | `Tainted size of variable length array`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Assertion

Failed assertion statement

Description

`Assertion` 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.

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.

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);
}
```

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 Comments”

Introduced in R2013b

Bad file access mode or status

Access mode argument of function in `fopen` or `open` group is invalid

Description

`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.

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.

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
CERT C ID: EXP37-C, FIO11-C

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Bad order of dropping privileges

Dropped higher elevated privileges before dropping lower elevated privileges

Description

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.

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.

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
CERT C ID: POS36-C

Introduced in R2016b

Base class assignment operator not called

Copy assignment operator does not call copy assignment operators of base subobjects

Description

`Base class assignment operator not called` 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 De nition`.
- 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 De nition`.

- 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 De nition.

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

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Copy constructor not called in initialization list`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Base class destructor not virtual

Class cannot behave polymorphically for deletion of derived class objects

Description

`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.

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.

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 Comments”

External Websites

CERT C++ OOP52-CPP

Introduced in R2015b

Bitwise and arithmetic operation on the same data

Statement with mixed bitwise and arithmetic operations

Description

`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.

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).

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
CERT C ID: INT14-C

Introduced in R2016b

Bitwise operation on negative value

Undefined behavior for bitwise operations on negative values

Description

`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.

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.

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
CERT C ID: INT13-C

Introduced in R2016b

Buffer overflow from incorrect string format specifier

String format specifier causes buffer argument of standard library functions to overflow

Description

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.

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.

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
CERT C ID: ARR38-C, STR03-C, STR31-C
ISO/IEC TS 17961 ID: taintformatio

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Call through non-prototyped function pointer

Function pointer declared without its type or number of parameters causes unexpected behavior

Description

`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.

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.

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
ISO/IEC TS 17961 ID: taintnoproto

See Also

Declaration mismatch | Unreliable cast of function pointer

Introduced in R2017b

Call to memset with unintended value

`memset` or `wmemset` used with possibly incorrect arguments

Description

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))`.

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.

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
CERT C ID: INT31-C

See Also

Polyspace Analysis Options
Find defects (-checkers)

Polyspace Results
Use of memset with size argument zero

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Character value absorbed into EOF

Data type conversion makes a valid character value same as End-of-File (EOF)

Description

`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.

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.

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
CERT C ID: FIO34-C
ISO/IEC TS 17961 ID: `chreof`

See Also

Polyspace Results
`Errno not checked` | `Invalid use of standard library integer routine` | `Misuse of sign-extended character value` | `Returned value of a sensitive function not checked`

Introduced in R2017a

Closing a previously closed resource

Function closes a previously closed stream

Description

`Closing a previously closed resource` 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.

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, 910
CERT C ID: FIO46-C

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Code deactivated by constant false condition

Code segment deactivated by `#if 0` directive or `if(0)` condition

Description

`Code deactivated by constant false condition` occurs when a block of code is deactivated using a `#if 0` directive or `if(0)` condition.

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.

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

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Dead code` | `Unreachable code` | `Useless if`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Command executed from externally controlled path

Path argument from an unsecure source

Description

`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.

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.

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
CERT C ID: API00-C, ENV33-C, STR02-C
ISO/IEC TS 17961 ID: syscall

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`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Constant block cipher initialization vector

Initialization vector is constant instead of randomized

Description

`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`.

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.

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
CERT C ID: MSC18-C

Introduced in R2017a

Constant cipher key

Encryption or decryption key is constant instead of randomized

Description

`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`.

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.

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
CERT C ID: MSC18-C

Introduced in R2017a

Context initialized incorrectly for cryptographic operation

Context used for public key cryptography operation is initialized for a different operation

Description

Context initialized incorrectly for cryptographic operation 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.

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

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

Introduced in R2018a

Context initialized incorrectly for digest operation

Context used for digest operation is initialized for a different digest operation

Description

`Context initialized incorrectly for digest operation` 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`.

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, 372, 573, 664

See Also

Nonsecure hash algorithm

Introduced in R2018a

Copy constructor not called in initialization list

Copy constructor does not call copy constructors of some members or base classes

Description

Copy constructor not called in initialization list 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 De nition. 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 De nition.

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

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Base class assignment operator not called`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Copy of overlapping memory

Source and destination arguments of a copy function have overlapping memory

Description

`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.

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
CERT C ID: EXP43-C, MSC15-C
ISO/IEC TS 17961 ID: `restrict`

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Overlapping assignment`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Data race

Multiple tasks perform unprotected non-atomic operations on shared variable

Description

Data race occurs when:

1. Multiple tasks perform unprotected operations on a shared variable.
2. At least one task performs a read operation and another task performs a write operation.
3. At least one operation is non-atomic. For data race on both atomic and non-atomic operations, see `Data race including atomic operations`.

A non-atomic operation can translate into more than one machine instruction. For instance:

- The operation can involve both a read and write operation. For example, `var++` involves reading the value of `var`, increasing the value by one and writing the increased value back to `var`.
- The operation can involve a 64-bit variable on a 32-bit target. For example, the operation

  ```
  long long var1, var2;
  var1=var2;
  ```

  involves two steps in copying the content of `var2` to `var1` on certain targets.

  Polyspace uses the `Pointer` size for your `Target processor type` as the threshold to compute atomicity. For instance, if you use `i386` as your `Target processor type`, the `Pointer` size is 32 bits, and `Long long` and `Double` sizes are both 64 bits. Therefore, Polyspace considers copying one `long long` or `double` variable to another as non-atomic.
- The operation can involve writing the return value of a function call to a shared variable. For example, the operation `x=func()` involves calling `func` and writing the return value of `func` to `x`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the Con guration 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 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 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 1-109	☑	
Tasks on page 1-116	task1 task2 task3	
Critical section details on page 1-128	Starting routine	Ending routine
	begin_critical_section	end_critical_section

On the command-line, you can use the following:

```
polyspace-bug-finder-nodesktop
  -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`.

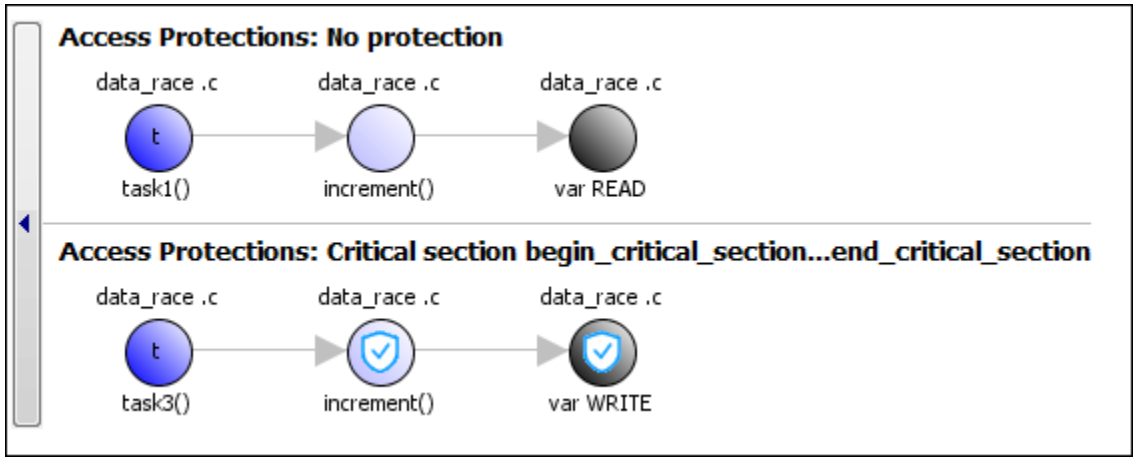
These machine instructions, when executed from `task1` and `task2`, can occur concurrently in an unpredictable sequence. For example, reading `var` from `task1` can occur either before or after writing to `var` from `task2`. Therefore the value of `var` can be unpredictable.

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`.



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();
    }
```

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the `Con guration` pane, specify the following additional options:

Option	Value
Temporally exclusive tasks on page 1-132	`task1 task2 task3`

On the command-line, you can use the following:

```
polyspace-code-prover-nodesktop
    -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

Check Information

Group: Concurrency
Language: C | C++
Default: On
Command-Line Syntax: DATA_RACE
Impact: High
CWE ID: 366
CERT C ID: CON09-C, CON32-C, CON43-C, POS49-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)` | `Target processor type (-target)` | `Configure multitasking manually` | `Tasks (-entry-points)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Temporally exclusive tasks (-temporal-exclusions-file)` | `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`

Polyspace Results
`Data race including atomic operations` | `Data race through standard library function call` | `Deadlock` | `Destruction of locked mutex` | `Double lock` | `Double unlock` | `Missing lock` | `Missing unlock`

Topics

“Analyze Multitasking Programs in Polyspace”
“Protections for Shared Variables in Multitasking Code”

Introduced in R2014b

Data race including atomic operations

Multiple tasks perform unprotected operations on shared variable

Description

Data race occurs when:

1. Multiple tasks perform unprotected operations on a shared variable.
2. At least one task performs a read operation and another 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`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the Con guration pane, select Multitasking.

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 on page 1-109	☑	

Tasks on page 1-116	`task1`
	`task2`
	`task3`	

Critical section details on page 1-128	Starting routine	Ending routine
	`begin_critical_section`	`end_critical_section`

On the command-line, you can use the following:

```
polyspace-bug-finder-nodesktop
  -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.

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();
}
```

Another possible correction is to make the tasks `task1` and `task2` temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the `Con guration` pane, specify the following additional options:

Option	Value
Temporally exclusive tasks on page 1-132	`task1 task2`

On the command-line, use the following:

```
polyspace-code-prover-nodesktop
    -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
CERT C ID: CON00-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)` | `Configure multitasking manually` | `Tasks (-entry-points)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Temporally exclusive tasks (-temporal-exclusions-file)` | `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`

Polyspace Results
`Data race` | `Data race through standard library function call` | `Deadlock` | `Destruction of locked mutex` | `Double lock` | `Double unlock` | `Missing lock` | `Missing unlock`

Topics

“Configuring Polyspace Multitasking Analysis Manually”

"Protections for Shared Variables 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

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.

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 on page 1-109	☑	

Tasks on page 1-116	`task1`
	`task2`
	`task3`	

Critical section details on page 1-128	Starting routine	Ending routine
	`begin_critical_section`	`end_critical_section`

On the command-line, you can use the following:

```
polyspace-bug-finder-nodesktop
  -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`.

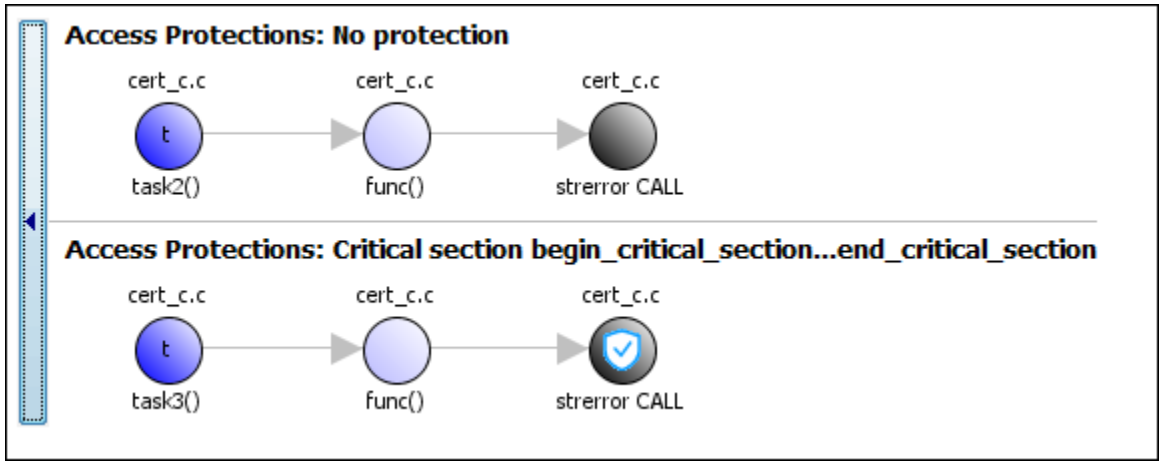
Though `task3` calls `func` inside a critical section, other tasks do not use the same critical section. Operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

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`.



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();
}
```

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();
}
```

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 1-132	`task1 task2 task3`

On the command-line, you can use the following:

```
polyspace-code-prover-nodesktop
    -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
CERT C ID: CON33-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)` | `Configure multitasking manually` | `Tasks (-entry-points)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Polyspace Results
`Data race including atomic operations` | `Data race` | `Destruction of locked mutex` | `Double lock` | `Double unlock` | `Missing lock` | `Missing unlock`

Topics

“Analyze Multitasking Programs in Polyspace”
“Protections for Shared Variables in Multitasking Code”

Introduced in R2016b

Deadlock

Call sequence to lock functions cause two tasks to block each other

Description

`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 `Con guration` pane, select `Multitasking`.

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	☑	
Entry points	task1 task2	
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`.

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
	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`.

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
CERT C ID: CON35-C, POS51-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)` | `Configure multitasking manually` | `Tasks (-entry-points)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Polyspace Results
`Data race including atomic operations` | `Data race` | `Data race through standard library function call` | `Destruction of locked mutex` | `Double lock` | `Double unlock` | `Missing lock` | `Missing unlock`

Topics

"Analyze Multitasking Programs in Polyspace"

Introduced in R2014b

Dead code

Code does not execute

Description

`Dead code` occurs when a block of code cannot be reached via any execution path. This defect excludes:

- `Code deactivated by constant false condition`, which checks for directives such as `#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.

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.

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.

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
CERT C ID: MSC01-C, MSC07-C, MSC12-C
ISO/IEC TS 17961 ID: `swtchdflt`

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Code deactivated by constant false condition` | `Unreachable code` | `Useless if`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Deallocation of previously deallocated pointer

Memory freed more than once without allocation

Description

`Deallocation of previously deallocated pointer` occurs when a block of memory is freed more than once using the `free` function without an intermediate allocation.

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.

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
CERT C ID: MEM00-C, MEM30-C
ISO/IEC TS 17961 ID: `accfree`, `dblfree`

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Use of previously freed pointer`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Declaration mismatch

Mismatch between function or variable declarations

Description

`Declaration mismatch` occurs when a function or variable declaration does not match other instances of the function or variable.

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`.

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;
}
```

le2.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.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.

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.

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
CERT C ID: DCL40-C, EXP37-C, MSC15-C
ISO/IEC TS 17961 ID: `argcomp`, `funcdecl`

See Also

Polyspace Analysis Options
`Find defects (-checkers)` | `Ignore pragma pack directives (-ignore-pragma-pack)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Delete of void pointer

`delete` operates on a `void*` pointer pointing to an object

Description

`Delete of void pointer` 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.

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 Comments”

Introduced in R2015b

Destination bu er over ow in string manipulation

Function writes to buffer at offset greater than buffer size

Description

Destination bu er over ow 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.

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.

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, 251, 787
CERT C ID: ARR38-C, STR07-C, STR31-C, STR38-C, ENV01-C
ISO/IEC TS 17961 ID: `libptr`, `taintformatio`

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Destination buffer underflow in string manipulation`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Destination buffer underflow in string manipulation

Function writes to buffer at a negative offset from beginning of buffer

Description

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.

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`.

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
CERT C ID: ARR38-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Destination buffer overflow in string manipulation`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Destruction of locked mutex

Task tries to destroy a mutex in the locked state

Description

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.

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`.

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.

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);
}
```

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
CERT C ID: CON31-C, POS48-C

See Also

Polyspace Analysis Options
Find defects (-checkers) | Target processor type (-target) | Configure multitasking manually | Tasks (-entry-points)

Polyspace Results

`Data race including atomic operations` | `Data race` | `Data race through standard library function call` | `Deadlock` | `Double lock` | `Double unlock` | `Missing lock` | `Missing unlock`

Topics

"Analyze Multitasking Programs in Polyspace"

Introduced in R2016b

Deterministic random output from constant seed

Seeding routine uses a constant seed making the output deterministic

Description

`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 3-727, 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.

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
CERT C ID: MSC32-C

See Also

`Predictable random output from predictable seed` | `Unsafe standard encryption function` | `Vulnerable pseudo-random number generator`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Double lock

Lock function is called twice in a task without an intermediate call to unlock function

Description

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 Con guration pane, select Multitasking.

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	
Con gure multitasking manually on page 1-109	☑	
Tasks on page 1-116	`task1` `task2`	
Critical section details on page 1-128	Starting routine	Ending routine
	`lock`	`unlock`

On the command-line, you can use the following:

```
polyspace-bug-finder-nodesktop
  -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();`.

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();
}
```

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();
}
```

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 on page 1-109	☑	
Tasks on page 1-116	`task1` `task2`	
Critical section details on page 1-128	Starting routine	Ending routine
	`lock`	`unlock`

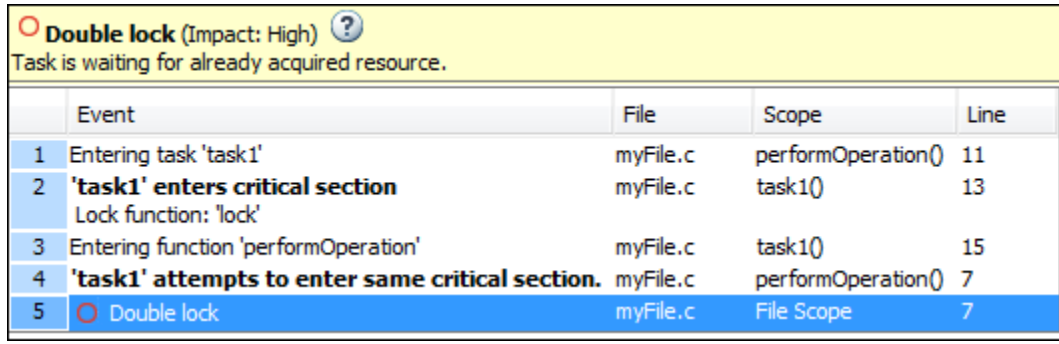
On the command-line, you can use the following:

```
polyspace-bug-finder-nodesktop
  -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.

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
CERT C ID: CON01-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)` | `Configure multitasking manually` | `Tasks (-entry-points)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Polyspace Results
`Data race including atomic operations` | `Data race` | `Data race through standard library function call` | `Deadlock` | `Destruction of locked mutex` | `Double unlock` | `Missing lock` | `Missing unlock`

Topics

"Analyze Multitasking Programs in Polyspace"

Introduced in R2014b

Double unlock

Unlock function is called twice in a task without an intermediate call to lock function

Description

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 Con guration pane, select Multitasking.

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 on page 1-109	☑	

Tasks on page 1-116	`task1`
	
	`task2`	

Critical section details on page 1-128	Starting routine	Ending routine
	`BEGIN_CRITICAL_SECTION`	`END_CRITICAL_SECTION`

On the command-line, you can use the following:

```
polyspace-bug-finder-nodesktop
  -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`.

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();
}
```

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();
}
```

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
CERT C ID: CON01-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)` | `Configure multitasking manually` | `Tasks (-entry-points)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Polyspace Results
`Data race including atomic operations` | `Data race` | `Data race through standard library function call` | `Deadlock` | `Destruction of locked mutex` | `Double lock` | `Missing lock` | `Missing unlock`

Topics
"Analyze Multitasking Programs in Polyspace"

Introduced in R2014b

Environment pointer invalidated by previous operation

Call to `setenv` or `putenv` family function modifies environment pointed to by pointer

Description

`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.

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.

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
CERT C ID: ENV31-C

See Also

Topics

Misuse of return value from nonreentrant standard function
Modification of internal buffer returned from nonreentrant standard function

Introduced in R2018a

Errno not checked

errno is not checked for error conditions following function call

Description

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.

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`.

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: 391
CERT C ID: ERR33-C
ISO/IEC TS 17961 ID: inverrno

See Also

Polyspace Results
Errno not reset | Misuse of errno | Returned value of a sensitive function not checked

Introduced in R2017a

Errno not reset

errno not reset before calling a function that sets errno

Description

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.

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.

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: 456, 703
CERT C ID: ERR30-C
ISO/IEC TS 17961 ID: `inverrno`

See Also

Polyspace Results
`Errno not reset` | `Errno not checked` | `Returned value of a sensitive function not checked`

Introduced in R2017a

Exception caught by value

`catch` statement accepts an object by value

Description

`Exception caught by value` 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.

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.

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 Comments”

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

`Exception handler hidden by previous handler` 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.

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 Comments”

Introduced in R2015b

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

Execution of a binary from a relative path can be controlled by an external actor detects calls to an external command. If the call uses a relative path or no path to call the external command, Bug Finder flags the call as a defect.

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. An attacker can manipulate the command to use a malicious version.

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

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Execution of externally controlled command

Command argument from an unsecure source vulnerable to operating system command injection

Description

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.

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.

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
CERT C ID: API00-C, ENV33-C, STR02-C
ISO/IEC TS 17961 ID: syscall

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

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

File access between time of check and use (TOCTOU)

File or folder might change state due to access race

Description

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.

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.

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
CERT C ID: FIO01-C, FIO45-C, POS35-C

See Also

`Data race` | `Bad file access mode or status`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

File descriptor exposure to child process

Copied file descriptor used in multiple processes

Description

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.

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.

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
CERT C ID: POS38-C

See Also

Introduced in R2017b

File manipulation after chroot() without chdir("/")

Path-related vulnerabilities for file manipulated after call to chroot

Description

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.

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.

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
CERT C ID: POS05-C

See Also

`Umask used with chmod-style arguments` | `Vulnerable path manipulation`

Topics

“Interpret Polyspace Bug Finder Results”

"Address Polyspace Results Through Bug Fixes or Comments"

Introduced in R2015b

Float conversion over ow

Overflow when converting between floating point data types

Description

Float conversion over ow 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)`.

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: 197, 681

CERT C ID: FLP03-C, FLP34-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Integer conversion overflow|Unsigned integer conversion overflow|Sign change integer conversion overflow`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Float division by zero

Dividing floating point number by zero

Description

`Float division by zero` occurs when the denominator of a division operation is a zero and a floating point number.

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.

```
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.

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: 369
CERT C ID: FLP03-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Integer division by zero`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Float over ow

Overflow from operation between floating points

Description

`Float over ow` occurs when an operation on floating point variables exceeds the space available to represent the resulting value.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

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.

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: 682, 873
CERT C ID: FLP03-C, FLP06-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Integer overflow` | `Unsigned integer overflow`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Format string specifiers and arguments mismatch

String specifiers do not match corresponding arguments

Description

Format string specifiers and arguments mismatch occurs when the parameters in the format specification do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

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`.

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);
}
```

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: 685, 686
CERT C ID: DCL10-C, DCL11-C, EXP37-C, FIO47-C, INT00-C, MSC15-C
ISO/IEC TS 17961 ID: `argcomp`, `invfmtstr`

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Invalid use of standard library string routine`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

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

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.

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.

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, 479, 663, 828
CERT C ID: SIG30-C
ISO/IEC TS 17961 ID: asyncsig

See Also

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

Introduced in R2017b

Function called from signal handler not asynchronous-safe (strict)

Call to interrupted function causes undefined program behavior

Description

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()

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.

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, 479, 663, 828
CERT C ID: SIG30-C
ISO/IEC TS 17961 ID: asyncsig

See Also

Function called from signal handler not asynchronous-safe | Shared data access within signal handler | Signal call from within signal handler

Introduced in R2017b

Function pointer assigned with absolute address

Constant expression is used as function address is vulnerable to code injection

Description

Function pointer assigned with absolute address looks for assignments to function pointers. If the function pointer is assigned an absolute address, Bug Finder raises a defect.

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 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.

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

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Hard-coded bu er size

Size of memory buffer is a numerical value instead of symbolic constant

Description

Hard-coded bu er 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.

Examples

Hard-Coded Bu er 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.

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
CERT C ID: DCL06-C

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Hard-coded loop boundary

Loop boundary is a numerical value instead of symbolic constant

Description

`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.

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.

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
CERT C ID: DCL06-C

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Hard-coded object size used to manipulate memory

Memory manipulation with hard-coded size instead of `sizeof`

Description

`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`*`)`.

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.

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
CERT C ID: EXP09-C

Introduced in R2016b

Host change using externally controlled elements

Changing host ID from an unsecure source

Description

`Host change using externally controlled elements` detects uncontrolled arguments in calls to routines that change the host ID, such as `sethostid` (Linux) or `SetComputerName` (Windows).

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.

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
CERT C ID: API00-C

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`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Improper array initialization

Incorrect array initialization when using initializers

Description

`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.

Issue	Risk	Possible Fix
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.

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}`.

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.

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.

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.

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
CERT C ID: ARR00-C, ARR02-C

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Incompatible padding for RSA algorithm operation

Cryptography operation is not supported by the padding type set in context

Description

`Incompatible padding for RSA algorithm operation` 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.

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

`Missing blinding for RSA algorithm` | `Missing padding for RSA algorithm` | `Nonsecure RSA public exponent` | `Weak padding for RSA algorithm`

Introduced in R2018a

Incompatible types prevent overriding

Derived class method hides a `virtual` base class method instead of overriding it

Description

`Incompatible types prevent overriding` occurs when a derived class method has the same name and number of parameters as a `virtual` base class method but:

- Differ in at least one parameter type.
- 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

Risks include the following:

- If you intend that the derived class method must override the base class method, the overriding does not occur.
- Because the base class method is hidden, you cannot use a derived class object to call the method. If 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

Possible solutions include the following:

- If you want the derived class method to override the base class method, change the interface of the derived class method.

  For instance, change the parameter type or add a `const` qualifier if required.
- Otherwise, add the line `using` *`Base_class_name`*`::`*`method_name`* to the derived class declaration. In this way, you can 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 De nition`.
2. In the base class definition, identify the `virtual` method that has the same name as the derived class method name.

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);
};
```

`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`.

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;

} ;
}
```

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`.

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) ;

} ;
}
```

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 Comments”

Introduced in R2015b

Inconsistent cipher operations

You perform encryption and decryption steps in succession with the same cipher context without a reinitialization in between

Description

`Inconsistent cipher operations` 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`.

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

Introduced in R2017a

Incorrect data type passed to va_arg

Data type of variadic function argument does not match type in `va_arg` call

Description

`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.

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.

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
CERT C ID: EXP47-C

See Also

`Invalid va_list argument` | `Too many va_arg calls for current argument list`

Introduced in R2018a

Incorrect key for cryptographic algorithm

Public key cryptography operation is not supported by the algorithm used in context initialization

Description

`Incorrect key for cryptographic algorithm` 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.

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 | 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

Introduced in R2018a

Incorrect order of network connection operations

Socket is not correctly established due to bad order of connection steps or missing steps

Description

`Incorrect order of network connection operations` 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.

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

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Incorrect pointer scaling

Implicit scaling in pointer arithmetic might be ignored

Description

`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.

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.

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.

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
CERT C ID: ARR39-C, EXP08-C
ISO/IEC TS 17961 ID: `libptr`

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Information leak via structure padding

Padding bytes can contain sensitive information

Description

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.

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`.

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
CERT C ID: DCL39-C

See Also

Topics

Memory comparison of padding data
Large pass-by-value argument
Use of memset with size argument zero
Invalid assumptions about memory organization
Sensitive heap memory not cleared before release
Uncleared sensitive data in stack

Introduced in R2018a

Inline constraint not respected

Modifiable static variable is modified in nonstatic inline function

Description

`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.

Examples

Static Variable Use in Inlined and External De nition

```
/* 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`.

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
CERT C ID: MSC40-C

Introduced in R2018a

Integer conversion overflow

Overflow when converting between integer types

Description

`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 constant, the conversion overflows.

The exact storage allocation for different integer types depends on your processor. See `Target processor type (-target)`.

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.

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, 190, 191, 192, 197
CERT C ID: FLP34-C, INT02-C, INT12-C, INT18-C, INT31-C

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 Comments”

Introduced in R2013b

Integer division by zero

Dividing integer number by zero

Description

`Integer division by zero` occurs when the denominator of a division or modulo operation is zero.

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.

```
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.

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.

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];
}
```

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: 369
CERT C ID: INT33-C
ISO/IEC TS 17961 ID: `diverr`

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
Float division by zero on page 3-201

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Integer over ow

Overflow from operation between integers

Description

`Integer over ow` occurs when an operation on integer variables exceeds the space available to represent the resulting value.

The exact storage allocation for different integer types depends on your processor. See `Target processor type (-target)`.

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.

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, 190, 191, 192
CERT C ID: INT00-C, INT08-C, INT18-C, INT32-C, MSC15-C
ISO/IEC TS 17961 ID: `intoflow`

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Unsigned integer overflow` | `Float overflow`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Invalid assumptions about memory organization

Address is computed by adding or subtracting from address of a variable

Description

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.

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.

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
CERT C ID: ARR37-C

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Invalid deletion of pointer

Pointer deallocation using `delete` without corresponding allocation using `new`

Description

`Invalid deletion of pointer` occurs when a block of memory released using the `delete` operator was not previously allocated with the `new` operator.

This defect applies only to C++ source files.

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.

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;
}
```

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.

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;
}
```

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

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Invalid free of pointer` | `Memory leak`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Invalid le position

`fsetpos()` is invoked with a file position argument not obtained from `fgetpos()`

Description

Invalid le 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()`.

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.

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
CERT C ID: FIO44-C
ISO/IEC TS 17961 ID: xfilepos

See Also

Introduced in R2017b

Invalid free of pointer

Pointer deallocation without a corresponding dynamic allocation

Description

`Invalid free of pointer` occurs when a block of memory released using the `free` function was not previously allocated using `malloc`, `calloc`, or `realloc`.

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.

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 */
 }
```

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
CERT C ID: MEM00-C, MEM34-C
ISO/IEC TS 17961 ID: `xfree`

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Invalid deletion of pointer`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Invalid use of = = operator

Equality operation in assignment statement

Description

`Invalid use of == operator` occurs when an equality operator instead of an assignment operator is used in a simple statement. A common correction is removing one of the equal signs (=).

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.

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

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Invalid use of = operator`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Invalid use of = operator

Assignment in conditional statement

Description

`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.

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.

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");
    }
}
```

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;
    }
}
```

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
CERT C ID: EXP45-C
ISO/IEC TS 17961 ID: `boolasgn`

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Invalid use of == operator`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Floating point comparison with equality operators

Imprecise comparison of floating-point variables

Description

Floating point comparison with equality operators occurs when you use an equality (==) or inequality (!=) operation with floating-point numbers. It is possible that the equality or inequality of two floating-point values is not exact because floating-point representation can be imprecise.

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; }
  ŭ
  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) {
  ŭ
  if (random) break;
  ```

```
}

if (3*x+4==2*x-1)
ŭ
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 Con guration pane. See `Other`.

  At the command line, add the flag to your analysis command.

  ```
  polyspace-bug-finder-nodesktop -sources filename ^
  -checkers BAD_FLOAT_OP -detect-bad-float-op-on-zero
  ```

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.

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
CERT C ID: FLP02-C

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Invalid use of standard library oating point routine

Wrong arguments to standard library function

Description

Invalid use of standard library oating 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`

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.

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
CERT C ID: FLP03-C, FLP32-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Invalid use of standard library integer routine | Invalid use of standard library memory routine | Invalid use of standard library string routine | Invalid use of standard library routine`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Invalid use of standard library integer routine

Wrong arguments to standard library function

Description

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`

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`.

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
CERT C ID: STR37-C
ISO/IEC TS 17961 ID: `chrsgnext`

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Invalid use of standard library floating point routine | Invalid use of standard library memory routine | Invalid use of standard library string routine | Invalid use of standard library routine`

Topics

“Interpret Polyspace Bug Finder Results”

"Address Polyspace Results Through Bug Fixes or Comments"

Introduced in R2013b

Invalid use of standard library memory routine

Standard library memory function called with invalid arguments

Description

Invalid use of standard library memory routine occurs when a memory library function is called with invalid arguments.

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.

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
CERT C ID: API00-C, ARR38-C, ARR39-C, EXP08-C, EXP34-C, MSC15-C
ISO/IEC TS 17961 ID: `nullref`, `libptr`

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Invalid use of standard library string routine`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Invalid use of standard library routine

Wrong arguments to standard library function

Description

`Invalid use of standard library routine` 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.

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.

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
CERT C ID: API00-C, MSC15-C
ISO/IEC TS 17961 ID: `strmod`

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Invalid use of standard library integer routine` | `Invalid use of standard library floating point 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 Comments"

Introduced in R2013b

Invalid use of standard library string routine

Standard library string function called with invalid arguments

Description

`Invalid use of standard library string routine` occurs when a string library function is called with invalid arguments.

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`.

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
CERT C ID: ARR38-C, STR31-C, STR32-C, MEM30-C, API00-C, MSC15-C
ISO/IEC TS 17961 ID: accfree, nullref, libptr, nonnullcs, taintformatio

See Also

Polyspace Analysis Options
Find defects (-checkers)

Polyspace Results
Invalid use of standard library memory routine

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

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

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.

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.

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
CERT C ID: MSC39-C

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 Comments”

Introduced in R2015b

Large pass-by-value argument

Large argument passed by value between functions

Description

`Large pass-by-value argument` occurs when a large input argument or return value is passed between functions by its value. For variables larger than 64 bytes, pass the value by pointer or by reference to save stack space and copy time.

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.

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.

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.

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 Comments”

Introduced in R2013b

Library loaded from externally controlled path

Using a library argument from an externally controlled path

Description

`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.

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.

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;
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: TAINTED_PATH_LIB
Impact: Medium
CWE ID: 114, 426
CERT C ID: API00-C, STR02-C, WIN00-C

See Also

Execution of externally controlled command | Use of externally controlled environment variable | Command executed from externally controlled path

Topics

"Interpret Polyspace Bug Finder Results"
"Address Polyspace Results Through Bug Fixes or Comments"

Introduced in R2015b

Line with more than one statement

Multiple statements on a line

Description

Before preprocessing starts, `Line with more than one statement` checks for additional text after the semicolon (`;`) on a line. A defect is not raised for comments, `for`-loop definitions, braces, or backslashes.

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.

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;
}
```

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.

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.

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 Comments”

Introduced in R2013b

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

`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.

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.

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
CERT C ID: WIN00-C

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`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Loop bounded with tainted value

Loop controlled by a value from an unsecure source

Description

`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.

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.

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
CERT C ID: API00-C, INT04-C, MSC21-C
ISO/IEC TS 17961 ID: `taintsink`

See Also

`Array access with tainted index` | `Pointer dereference with tainted offset`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Member not initialized in constructor

Constructor does not initialize some members of a class

Description

Non-initialized member 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 De nition. In the class definition, you can see all the members, including those members that are not initialized in the constructor.

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
ISO/IEC TS 17961 ID: uninitref

See Also

Polyspace Analysis Options
Find defects (-checkers)

Polyspace Results
Copy constructor not called in initialization list

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Memory allocation with tainted size

Size argument to memory function is from an unsecure source

Description

`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.

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.

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
CERT C ID: API00-C, ARR32-C, INT04-C, MEM07-C, MEM10-C, MEM11-C, MEM35-C
ISO/IEC TS 17961 ID: `taintsink`

See Also

`Unprotected dynamic memory allocation`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Memory comparison of oat-point values

Object representation of floating-point values can be different (same) for equal (not equal) floating-point values

Description

Memory comparison of oat-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.

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.

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);
}
```

Result Information

Group: Programming
Language: C | C++
Default: Off
Command-Line Syntax: MEMCMP_FLOAT
Impact: Low
CERT C ID: FLP37-C

See Also

Topics

Floating point comparison with equality operators
Memory comparison of padding data
Memory comparison of strings

Introduced in R2018a

Memory comparison of padding data

`memcmp` compares data stored in structure padding

Description

`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.

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.

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
CERT C ID: EXP42-C
ISO/IEC TS 17961 ID: padcomp

See Also

Polyspace Results
Memory comparison of strings

Introduced in R2017a

Memory comparison of strings

`memcmp` compares data stored in strings after the null terminator

Description

`Memory comparison of strings` 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.

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

Polyspace Results
`Memory comparison of padding data`

Introduced in R2017a

Memory leak

Memory allocated dynamically not freed

Description

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.

Examples

Pointer with Dynamic Memory

```
#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`.

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);
}
```

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.

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
CERT C ID: MEM11-C, MEM12-C, MEM31-C
ISO/IEC TS 17961 ID: fileclose

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Mismatched alloc/dealloc functions on Windows

Improper deallocation function causes memory corruption issues

Description

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()`

Allocation Function	Deallocation Function
_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.

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
CERT C ID: WIN30-C

See Also

`Invalid deletion of pointer` | `Invalid free of pointer`

Introduced in R2017b

Mismatch between data length and size

Data size argument is not computed from actual data length

Description

`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.

Examples

Copy Bu er 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`.

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
CERT C ID: ARR38-C

See Also

Copy of overlapping memory

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Missing blinding for RSA algorithm

Context used in decryption or signature verification is not blinded against timing attacks

Description

Missing blinding for RSA algorithm 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.

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

Incompatible padding for RSA algorithm operation | Missing padding for RSA algorithm | Nonsecure RSA public exponent | Weak padding for RSA algorithm

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

`Missing block cipher initialization vector` 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.

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

Introduced in R2017a

Missing break of switch case

No comments at the end of switch case without a break statement

Description

`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.

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.

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
CERT C ID: MSC17-C

See Also

Missing case for switch condition

Introduced in R2016b

Missing byte reordering when transferring data

Different endianness of host and network

Description

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()` .

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.

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
CERT C ID: POS39-C

See Also

Introduced in R2017b

Missing case for switch condition

`switch` variable not covered by cases and default case is missing

Description

`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.

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.

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
CERT C ID: MSC01-C, MSC07-C
ISO/IEC TS 17961 ID: `swtchdflt`

See Also

`Find defects (-checkers)`

Topics

"Interpret Polyspace Bug Finder Results"
"Address Polyspace Results Through Bug Fixes or Comments"

Introduced in R2015b

Missing cipher algorithm

An encryption or decryption algorithm is not associated with the cipher context

Description

`Missing cipher algorithm` 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.

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

Introduced in R2017a

Missing cipher data to process

Final encryption or decryption step is performed without previous update steps

Description

Missing cipher data to process 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.

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

Introduced in R2017a

Missing cipher nal step

You do not perform a final step after update steps for encrypting or decrypting data

Description

Missing cipher nal step 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.

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

Introduced in R2017a

Missing cipher key

Context used for encryption or decryption is associated with NULL key or not associated with a key

Description

`Missing cipher key` 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.

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

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

`Missing data for encryption, decryption or signing operation` 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.

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 | Incorrect key for cryptographic algorithm | Missing parameters for key generation | Missing peer key | Missing private key | Missing public key | Nonsecure parameters for key generation

Introduced in R2018a

Missing explicit keyword

Constructor missing the `explicit` specifier

Description

`Missing explicit keyword` occurs when the declaration of a constructor 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){}`.

Risk

If you do not declare a constructor `explicit`, compilers can perform unexpected and often unintended type conversions to the class type using the constructor.

The implicit conversion 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.

Fix

For better readability of your code and to prevent implicit conversions, in the constructor declaration, place the `explicit` keyword before the constructor name.

If you want to convert from a variable of another type, explicitly call the class constructor and pass the variable as argument.

Examples

Missing `explicit` Keyword

```
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`.

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.

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
}
```

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 Comments”

Introduced in R2015b

Missing lock

Unlock function without lock function

Description

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 Con guration pane, select Multitasking.

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 on page 1-109	☑	
Tasks on page 1-116	`my_task`, `reset`	
Critical section details on page 1-128	Starting routine	Ending routine
	`begin_critical_section`	`end_critical_section`

On the command-line, you can use the following:

```
polyspace-bug-finder-nodesktop
   -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`.

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 on page 1-109	☑	
Tasks on page 1-116	`my_task`, `reset`	
Critical section details on page 1-128	Starting routine	Ending routine
	`begin_critical_section`	`end_critical_section`

On the command-line, you can use the following:

```
polyspace-bug-finder-nodesktop
   -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
CERT C ID: CON01-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)` | `Configure multitasking manually` | `Tasks (-entry-points)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Polyspace Results
`Data race including atomic operations` | `Data race` | `Data race through standard library function call` | `Deadlock` | `Destruction of locked mutex` | `Double lock` | `Double unlock` | `Missing unlock`

Topics
"Configuring Polyspace Multitasking Analysis Manually"

Introduced in R2014b

Missing null in string array

String does not terminate with null character

Description

`Missing null in string array` occurs when a string does not have enough space to terminate with a null character `'\0'`. This defect can cause various memory errors in your code, so is important to fix it.

This defect applies only for projects in C.

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.

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";
}
```

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
CERT C ID: STR11-C, STR31-C
ISO/IEC TS 17961 ID: NONNULLCS, TAINTFORMATIO

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Missing padding for RSA algorithm

Context used in encryption or signing operation is not associated with any padding

Description

`Missing padding for RSA algorithm` 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`.

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

Incompatible padding for RSA algorithm operation | Missing blinding for RSA algorithm | Nonsecure RSA public exponent | Weak padding for RSA algorithm

Introduced in R2018a

Missing parameters for key generation

Context used for key generation is associated with NULL parameters

Description

`Missing parameters for key generation` 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.

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 | 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

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

`Missing peer key` 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.

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 | 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

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

`Missing private key` 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.

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 | 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

Introduced in R2018a

Missing public key

Context used for cryptography operation is associated with NULL public key or not associated with a public key at all

Description

`Missing public key` 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.

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 | 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

Introduced in R2018a

Missing reset of a freed pointer

Pointer `free` not followed by a reset statement to clear leftover data

Description

`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.

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.

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
CERT C ID: MEM01-C

See Also

`Use of previously freed pointer` | `Invalid free of pointer`

Introduced in R2016b

Missing return statement

Function does not return value though return type is not `void`

Description

`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.

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.

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
CERT C ID: MSC37-C

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Missing unlock

Lock function without unlock function

Description

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.

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	
Con gure multitasking manually on page 1-109	☑	
Tasks on page 1-116	`my_task`, `reset`	
Critical section details on page 1-128	Starting routine	Ending routine
	`begin_critical_sectio n`	`end_critical_section`

On the command-line, you can use the following:

```
polyspace-bug-finder-nodesktop
   -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`.

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 on page 1-109	☑	
Tasks on page 1-116	`my_task`, `reset`	
Critical section details on page 1-128	Starting routine	Ending routine
	`begin_critical_section`	`end_critical_section`

On the command-line, you can use the following:

```
polyspace-bug-finder-nodesktop
   -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`.

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++;
    }
}
```

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
CERT C ID: CON01-C, MEM12-C

See Also

Polyspace Analysis Options
Find defects (-checkers) | Configure multitasking manually | Tasks (-entry-points) | Critical section details (-critical-section-begin -critical-section-end) | Temporally exclusive tasks (-temporal-exclusions-file)

Polyspace Results
Data race including atomic operations | Data race | Data race through standard library function call | Deadlock | Destruction of locked mutex | Double lock | Double unlock | Missing lock

Topics

“Configuring Polyspace Multitasking Analysis Manually”

Introduced in R2014b

Missing virtual inheritance

A base class is inherited virtually and nonvirtually in the same hierarchy

Description

Missing virtual inheritance 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 De nition.
- 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 De nition.

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 Comments”

Introduced in R2015b

Misuse of a FILE object

Use of copy of FILE object

Description

`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.

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.

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
CERT C ID: FIO38-C
ISO/IEC TS 17961 ID: `filecpy`

See Also

Introduced in R2017b

Misuse of structure with flexible array member

Memory allocation ignores flexible array member

Description

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.

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.

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
CERT C ID: MEM33-C

See Also

Introduced in R2017b

Misuse of errno

`errno` incorrectly checked for error conditions

Description

`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.

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.

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
CERT C ID: ERR30-C
ISO/IEC TS 17961 ID: inverrno

See Also

Polyspace Results
Errno not reset | Errno not checked | Returned value of a sensitive function not checked | Unsafe conversion from string to numerical value

Introduced in R2017a

Misuse of errno in a signal handler

You read `errno` after calling an `errno`-setting function in a signal handler

Description

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.

Examples

Reading `errno` After `signal` Call in Signal Handler

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

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`.

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
CERT C ID: ERR32-C

See Also

Errno not checked | Errno not reset | Function called from signal handler not asynchronous-safe

Introduced in R2018a

Misuse of readlink()

Third argument of `readlink` does not leave space for null terminator in buffer

Description

`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.

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';
```

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
CERT C ID: POS30-C

See Also

Polyspace Results
Array access out of bounds | File access between time of check and use (TOCTOU) | Invalid use of standard library string routine | Pointer access out of bounds | Returned value of a sensitive function not checked

Introduced in R2017a

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

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.

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.

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
CERT C ID: ENV34-C
ISO/IEC TS 17961 ID: libuse

See Also

Polyspace Results
Modification of internal buffer returned from nonreentrant standard function | Use of obsolete standard function

Introduced in R2017a

Misuse of sign-extended character value

Data type conversion with sign extension causes unexpected behavior

Description

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. Then you use the resulting sign-extended value as array index or for comparison with EOF.

Risk

Comparison with EOF: Suppose, your compiler implements the plain `char` type as signed. On 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, sign-extended plain `char` variables cannot be used 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.

Fix

Cast the signed or plain `char` value explicitly to `unsigned char` before conversion to a wider integer data type.

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 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.

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();
    }
}
```

Result Information

Group: Programming
Language: C | C++
Default: On
Command-Line Syntax: CHARACTER_MISUSE
Impact: Medium
CWE ID: 704
CERT C ID: STR34-C, STR37-C
ISO/IEC TS 17961 ID: signconv

See Also

Polyspace Results
Character value absorbed into EOF | Errno not checked | Invalid use of standard library integer routine | Returned value of a sensitive function not checked

Introduced in R2017a

Modi cation of internal bu er returned from nonreentrant standard function

Function attempts to modify internal buffer returned from a nonreentrant standard function

Description

Modi cation of internal bu er 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.

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.

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
CERT C ID: ENV30-C, STR06-C
ISO/IEC TS 17961 ID: libmod

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Non-initialized pointer

Pointer not initialized before dereference

Description

`Non-initialized pointer` occurs when a pointer is not assigned an address before dereference.

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.

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
CERT C ID: EXP33-C, MSC15-C
ISO/IEC TS 17961 ID: uninitref

See Also

Polyspace Analysis Options
Find defects (-checkers)

Polyspace Results
Non-initialized variable

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Non-initialized variable

Variable not initialized before use

Description

`Non-initialized variable` occurs when a variable is not initialized before its value is read.

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.

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
CERT C ID: EXP33-C, MSC15-C, MSC39-C
ISO/IEC TS 17961 ID: `uninitref`

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Non-initialized pointer`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Nonsecure hash algorithm

Context used for message digest creation is associated with weak algorithm

Description

`Nonsecure hash algorithm` 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`.

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

See Also

Context initialized incorrectly for digest operation

Introduced in R2018a

Nonsecure parameters for key generation

Context used for key generation is associated with weak parameters

Description

Nonsecure parameters for key generation 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

Insu cient 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.

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

See Also

Context initialized incorrectly for cryptographic operation | 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`

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

Nonsecure RSA public exponent 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.

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

See Also

Incompatible padding for RSA algorithm operation|Missing blinding for RSA algorithm|Missing padding for RSA algorithm|Weak padding for RSA algorithm

Introduced in R2018a

Nonsecure SSL/TLS protocol

Context used for handling SSL/TLS connections is associated with weak protocol

Description

Nonsecure SSL/TLS protocol 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.

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, 693

Introduced in R2018a

Null pointer

NULL pointer dereferenced

Description

Null pointer occurs when you use a pointer with a value of NULL as if it points to a valid memory location.

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.

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
CERT C ID: EXP34-C, MSC15-C
ISO/IEC TS 17961 ID: nullref

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Arithmetic operation with NULL pointer` | `Non-initialized pointer`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Object slicing

Derived class object passed by value to function with base class parameter

Description

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.

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()`.

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 Comments”

Introduced in R2015b

Opening previously opened resource

Opening an already opened file

Description

`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.

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.

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, 675
CERT C ID: FIO24-C

Introduced in R2016b

Overlapping assignment

Memory overlap between left and right sides of an assignment

Description

`Overlapping assignment` 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
CERT C ID: MSC15-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Copy of overlapping memory`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Partial override of overloaded virtual functions

Class overrides fraction of inherited virtual functions with a given name

Description

Partial override of overloaded virtual functions 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 De nition.
2. In the base class definition, identify the method that has the same name and signature as a derived class method name.

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 Comments”

Introduced in R2015b

Partially accessed array

Array partly read or written before end of scope

Description

`Partially accessed array` 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.

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.

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 Comments”

Introduced in R2013b

Pointer access out of bounds

Pointer dereferenced outside its bounds

Description

`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.

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.

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
CERT C ID: API02-C, ARR30-C, ARR38-C, ARR39-C, EXP08-C, EXP39-C, MEM35-C, MSC15-CSTR31-C
ISO/IEC TS 17961 ID: `ptrcomp`, `insufmem`, `invptr`, `taintformatio`

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Array access out of bounds`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Pointer dereference with tainted o set

Offset is from an unsecure source and dereference may be out of bounds

Description

Pointer dereference with tainted o set 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.

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.

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
CERT C ID: API00-C, API02-C, ARR30-C
ISO/IEC TS 17961 ID: invptr

See Also

Array access with tainted index | Use of tainted pointer

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Pointer or reference to stack variable leaving scope

Pointer to local variable leaves the variable scope

Description

`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.

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`,

Result Information

Group: Static memory
Language: C | C++
Default: On
Command-Line Syntax: LOCAL_ADDR_ESCAPE
Impact: High
CWE ID: 562
CERT C ID: DCL30-C
ISO/IEC TS 17961 ID: addrescape

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Pointer to non-initialized value converted to const pointer

Pointer to constant assigned address that does not contain a value

Description

Pointer to non initialized value converted to const pointer occurs when a pointer to a constant is assigned an address that does not yet contain a value.

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.

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
ISO/IEC TS 17961 ID: `uninitref`

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”

"Address Polyspace Results Through Bug Fixes or Comments"

Introduced in R2013b

Possible misuse of sizeof

Use of `sizeof` operator can cause unintended results

Description

`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.

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.

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
CERT C ID: ARR00-C, ARR01-C, ARR38-C, ARR39-C
ISO/IEC TS 17961 ID: `libptr`, `insufmem`, `sizeofptr`

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

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

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.

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.

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
CERT C ID: EXP00-C, EXP13-C

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

External Websites

C++ Operator Precedence

Introduced in R2015b

Prede ned macro used as an object

You use standard library macros such as `assert` and `errno` as objects

Description

`Prede ned 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.

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.

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
CERT C ID: MSC38-C

See Also

MISRA C:2012 Rule 21.2

Introduced in R2018a

Predictable block cipher initialization vector

Initialization vector is generated from a weak random number generator

Description

`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`.

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.

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
CERT C ID: MSC18-C

Introduced in R2017a

Predictable cipher key

Encryption or decryption key is generated from a weak random number generator

Description

`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`.

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.

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
CERT C ID: MSC18-C

Introduced in R2017a

Predictable random output from predictable seed

Seeding routine uses a predictable seed making the output predictable

Description

`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 3-727, 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`.

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
CERT C ID: MSC32-C

See Also

`Deterministic random output from constant seed` | `Unsafe standard encryption function` | `Vulnerable pseudo-random number generator`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Preprocessor directive in macro argument

You use a preprocessor directive in the argument to a function-like macro

Description

`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
```

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()`.

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
CERT C ID: PRE32-C

See Also

MISRA C:2012 Rule 20.6

Introduced in R2018a

Privilege drop not verified

Verify privilege relinquishment was successful

Description

`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.

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.

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
CERT C ID: POS37-C

Introduced in R2016b

Quali er removed in conversion

Variable qualifier is lost during conversion

Description

Quali er removed in conversion occurs during a conversion when one variable 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.

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.

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);
}
```

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
CERT C ID: EXP05-C, EXP32-C, EXP37-C
ISO/IEC TS 17961 ID: `argcomp`

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Resource leak

File stream not closed before `FILE` pointer scope ends or pointer is reassigned

Description

`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.

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`.

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
CERT C ID: FIO42-C, MEM12-C
ISO/IEC TS 17961 ID: `fileclose`

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Return from computational exception signal handler

Undefined behavior when signal handler returns normally from program error

Description

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.

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.

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
CERT C ID: SIG35-C

See Also

Function called from signal handler not asynchronous-safe | Function called from signal handler not asynchronous-safe (strict) | Signal call from within signal handler

Introduced in R2017b

Return of non const handle to encapsulated data member

Method returns pointer or reference to internal member of object

Description

`Return of non-const handle to encapsulated data member` 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`.

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

Polyspace Analysis Options
`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Returned value of a sensitive function not checked

Sensitive functions called without checking for unexpected return values and errors

Description

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.

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.

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);
}
```

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.

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, 690, 754
CERT C ID: EXP12-C, ERR33-C, POS54-C
ISO/IEC TS 17961 ID: liberr

Introduced in R2016b

Self assignment not tested in operator

Copy assignment operator does not test for self-assignment

Description

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.

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.

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 Comments”

Introduced in R2015b

Sensitive data printed out

Function prints sensitive data

Description

`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.

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`.

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
CERT C ID: MEM06-C

See Also

`Sensitive heap memory not cleared before release` | `Uncleared sensitive data in stack`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Sensitive heap memory not cleared before release

Sensitive data not cleared or released by memory routine

Description

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`.

Examples

Sensitive Bu er 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.

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
CERT C ID: MEM03-C, MSC18-C

See Also

`Uncleared sensitive data in stack` | `Sensitive data printed out`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Shared data access within signal handler

Access or modification of shared data causes inconsistent state

Description

`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`.

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.

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
CERT C ID: SIG31-C
ISO/IEC TS 17961 ID: accsig

See Also

Function called from signal handler not asynchronous-safe | Signal call from within signal handler

Introduced in R2017b

Shift of a negative value

Shift operator on negative value

Description

`Shift of a negative value` occurs when a bit-wise shift is used on a negative number. Shifts can overwrite the sign bit that identifies a number as negative.

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.

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
CERT C ID: INT34-C

See Also

Polyspace Analysis Options
Find defects (-checkers)

Polyspace Results
Shift operation overflow

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Shift operation over ow

Overflow from shifting operation

Description

Shift operation over ow occurs when a shift operation exceeds the space available to represent the resulting value.

The exact storage allocation for different data types depends on your processor. See `Target processor type (-target)`.

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.

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: 190
CERT C ID: INT34-C

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Side e ect of expression ignored

`sizeof`, `_Alignof`, or `_Generic` operates on expression with side effect

Description

Side e ect 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.

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`.

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

CERT C ID: EXP44-C

See Also

MISRA C:2012 Rule 13.6

Introduced in R2018a

Sign change integer conversion over ow

Overflow when converting between signed and unsigned integers

Description

`Sign change integer conversion over ow` 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 integer types depends on your processor. See `Target processor type (-target)`.

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.

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
CERT C ID: INT31-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Float conversion overflow | Unsigned integer conversion overflow | Integer conversion overflow`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Signal call from within signal handler

Nonpersistent signal handler calling `signal()` in Windows system causes race condition

Description

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.

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.

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
CERT C ID: SIG34-C
ISO/IEC TS 17961 ID: sigcall

See Also

Topics

Function called from signal handler not asynchronous-safe
Return from computational exception signal handler
Shared data access within signal handler

Introduced in R2017b

Standard function call with incorrect arguments

Argument to a standard function does not meet requirements for use in the function

Description

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.

Function Type	Situation	Risk	Fix
	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.
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.

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).

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
CERT C ID: EXP37-C, STR32-C, FIO46-C, API00-C, MSC15-C
ISO/IEC TS 17961 ID: argcomp, liberr, nonnullcs

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Static uncalled function

Function with static scope not called in file

Description

Static uncalled function occurs when a `static` function is not called in the same file where it is defined.

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`.

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 Comments”

Introduced in R2013b

Stream argument with possibly unintended side effects

Stream argument side effects occur more than once

Description

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.

Examples

Stream Argument of `getc()` Has Side E ect `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.

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
CERT C ID: FIO41-C

See Also

Topics

Returned value of a sensitive function not checked
Opening previously opened resource
Standard function call with incorrect arguments

Introduced in R2018a

Subtraction or comparison between pointers to di erent arrays

Subtraction or comparison between pointers causes undefined behavior

Description

Subtraction or comparison between pointers to di erent 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.

Examples

Subtraction Between Pointers to Elements in Di erent 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.

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
CERT C ID: ARR36-C
ISO/IEC TS 17961 ID: `ptrobj`

See Also

Introduced in R2017b

Tainted division operand

Division / operands from an unsecure source

Description

`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.

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.

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: 190, 369
CERT C ID: API00-C, INT32-C, INT33-C
ISO/IEC TS 17961 ID: `diverr`

See Also

`Integer division by zero` | `Float division by zero` | `Tainted modulo operand`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Tainted modulo operand

Remainder % operands are from an unsecure source

Description

`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.

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.

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
CERT C ID: API00-C, INT10-C, INT32-C, INT33-C
ISO/IEC TS 17961 ID: `diverr`, `intoflow`

See Also

`Integer division by zero` | `Tainted division operand`

Topics

“Interpret Polyspace Bug Finder Results”

"Address Polyspace Results Through Bug Fixes or Comments"

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

`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.

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.

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);
}
```

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
CERT C ID: STR31-C, STR32-C, FIO17-C, ENV01-C, API00-C
ISO/IEC TS 17961 ID: `nonnullcs`, `taintstrcpy`, `taintformatio`

See Also

`Tainted string format`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Tainted sign change conversion

Value from an unsecure source changes sign

Description

`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.

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.

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
CERT C ID: API00-C, INT02-C, INT31-C, MEM04-C, MEM11-C, MSC21-C
ISO/IEC TS 17961 ID: taintsink

See Also

Sign change integer conversion overflow

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

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

`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.

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.

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
CERT C ID: API00-C, ARR32-C, INT04-C, MEM04-C, MEM05-C
ISO/IEC TS 17961 ID: `taintsink`

See Also

`Memory allocation with tainted size`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Tainted string format

Input format argument is from an unsecure source

Description

`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.

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.

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
CERT C ID: API00-C, FIO30-C
ISO/IEC TS 17961 ID: `usrfmt`

See Also

`Tainted NULL or non-null-terminated string`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Too many va_arg calls for current argument list

Number of calls to `va_arg` exceeds number of arguments passed to variadic function

Description

`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.

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.

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
CERT C ID: EXP47-C

See Also

Topics

Invalid va_list argument
Incorrect data type passed to va_arg

Introduced in R2018a

Typedef mismatch

Mismatch between `typedef` statements

Description

`Typedef mismatch` detects `typedef` statements with different underlying types for these fundamental 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

```
typedef unsigned char size_t;

void func2()
{
    size_t var = 0;
    /*... more code ... */
}
```

file2.c

```
#include <stddef.h>

void func1()
{
    size_t var = 0;
    /*... more code ... */
}
```

In this example, Polyspace flags the definition of `size_t` in `file1.c` as a defect. This definition is a `typedef` mismatch because another file in your project, `file2.c`, includes `stddef.h`, which defines `size_t` as unsigned long.

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 func2()
{
    size_t var = 0;
    /*... more code ... */
}
```

file2.c

```
#include <stddef.h>

void func1()
{
    size_t var = 0;
    /*... more code ... */
}
```

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 func2()
{
    size_t var = 0;
    /*... more code ... */
}
```

file2.c

```
#include "types.h"

void func1()
{
    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

Introduced in R2016b

Umask used with chmod-style arguments

Argument to umask allows external user too much control

Description

Umask used with chmod-style arguments checks for umask commands that have an argument 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.

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
CERT C ID: FIO06-C

See Also

Vulnerable permission assignments

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

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

`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`.

Examples

Static Bu er 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.

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
CERT C ID: MEM03-C, MSC18-C

See Also

Sensitive heap memory not cleared before release | Sensitive data printed out

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Universal character name from token concatenation

You create a universal character name by joining tokens with ## operator

Description

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.

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`.

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
CERT C ID: PRE30-C

See Also

`MISRA C:2012 Rule 20.10`

Introduced in R2018a

Unprotected dynamic memory allocation

Pointer returned from dynamic allocation not checked for `NULL` value

Description

`Unprotected dynamic memory allocation` occurs when the code does not check whether or not the dynamic memory allocation succeeded.

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 the `NULL` value, this access is not protected from failures.

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`

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: Dynamic memory
Language: C | C++
Default: Off
Command-Line Syntax: UNPROTECTED_MEMORY_ALLOCATION
Impact: Low
CWE ID: 690, 789
CERT C ID: MEM10-C, MEM11-C, ERR33-C
ISO/IEC TS 17961 ID: liberr

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Unreachable code

Code following control-flow statements

Description

`Unreachable code` defects occur on code which cannot be reached because the preceding code.

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.

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;
}
```

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;
}
```

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.

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;
}
```

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;
}
```

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
CERT C ID: MSC01-C, MSC07-C, MSC12-C
ISO/IEC TS 17961 ID: swtchdflt

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Code deactivated by constant false condition | Dead code | Useless if`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Unreliable cast of function pointer

Function pointer cast to another function pointer with different argument or return type

Description

`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.

Examples

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)`.

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

Group: Static memory
Language: C/C++
Default: On
Command-Line Syntax: FUNC_CAST
Impact: Medium
CERT C ID: EXP37-C, MSC15-C
ISO/IEC TS 17961 ID: argcomptaintnoproto

See Also

Polyspace Analysis Options
Find defects (-checkers)

Polyspace Results
Unreliable cast of pointer

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Unreliable cast of pointer

Pointer implicitly cast to different data type

Description

`Unreliable cast of pointer` 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.

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*`.

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: 704, 843
CERT C ID: EXP36-C, EXP39-C, MSC15-C, STR38-C
ISO/IEC TS 17961 ID: `alignconv`, `ptrcomp`

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Unreliable cast of function pointer`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Unsafe call to a system function

Unsanitized command argument has exploitable vulnerabilities

Description

`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()`.

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.

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
CERT C ID: ENV33-C
ISO/IEC TS 17961 ID: syscall

See Also

Command executed from externally controlled path | Execution of externally controlled command

Introduced in R2017b

Unsafe conversion between pointer and integer

Misaligned or invalid results from conversions between pointer and integer types

Description

`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`

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.

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
CERT C ID: INT36-C

ISO/IEC TS 17961 ID: `intptrconv`

Introduced in R2016b

Unsafe conversion from string to numerical value

String to number conversion without validation checks

Description

`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`.

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.

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, 676
CERT C ID: ERR34-C

Introduced in R2016b

Unsafe standard encryption function

Function is not reentrant or uses a risky encryption algorithm

Description

`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.

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.

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, 663
CERT C ID: MSC18-C

See Also

Deterministic random output from constant seed | Predictable random output from predictable seed | Vulnerable pseudo-random number generator

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Unsafe standard function

Function unsafe for security-related purposes

Description

`Unsafe standard function` looks for 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 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.

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`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Unsigned integer conversion over ow

Overflow when converting between unsigned integer types

Description

`Unsigned integer conversion over ow` 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 integer types depends on your processor. See `Target processor type (-target)`.

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`.

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, 190, 191, 192, 197
CERT C ID: FLP34-C, INT02-C, INT18-C, INT31-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Float conversion overflow | Integer conversion overflow | Sign change integer conversion overflow`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Unsigned integer over ow

Overflow from operation between unsigned integers

Description

Unsigned integer over ow occurs when an operation on unsigned integer variables exceeds the space available to represent the resulting value. The exact storage allocation for different integer types depends on your processor. See `Target processor type (-target)`.

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`.

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, 190, 191, 192
CERT C ID: INT18-C, INT30-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Integer overflow` | `Float overflow`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Unused parameter

Function prototype has parameters not read or written in function body

Description

`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.

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`.

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;
}
```

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
CERT C ID: MSC13-C

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Useless if

Unnecessary if conditional

Description

`Useless if` 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.

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.

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);
    }
}
```

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

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Code deactivated by constant false condition` | `Dead code` | `Unreachable code`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Use of automatic variable as putenv-family function argument

putenv-family function argument not accessible outside its scope

Description

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.

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.

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 */
    }
}
```

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
CERT C ID: POS34-C
ISO/IEC TS 17961 ID: addrescape

See Also

Pointer or reference to stack variable leaving scope

Introduced in R2017b

Use of dangerous standard function

Dangerous functions cause possible buffer overflow in destination buffer

Description

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.

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.

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
CERT C ID: PRE09-C, STR07-C, STR31-C, ENV01-C, API02-C
ISO/IEC TS 17961 ID: taintformatio

See Also

Use of obsolete standard function | Unsafe standard function | Invalid use of standard library string routine

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Use of externally controlled environment variable

Value of environment variable from an unsecure source

Description

Use of externally controlled environment variable checks for functions that add or change environment variables, such as `putenv` and `setenv`. If the new environment variable value is from an unsecure source, Polyspace raises a defect on the function or function pointer.

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.

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
CERT C ID: API00-C

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`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Use of indeterminate string

Use of buffer from fgets-family function

Description

`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.

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.

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
CERT C ID: FIO40-C

See Also

`Invalid use of standard library string routine` | `Returned value of a sensitive function not checked` | `Use of dangerous standard function`

Introduced in R2017b

Use of memset with size argument zero

Size argument of function in `memset` family is zero

Description

`Use of memset with size argument zero` 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
CERT C ID: MSC12-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`Call to memset with unintended value`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Use of non-secure temporary file

Temporary generated file name not secure

Description

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.

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 3-688.

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
CERT C ID: FIO03-C, FIO21-C

See Also

`Data race`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Use of obsolete standard function

Obsolete routines can cause security vulnerabilities and portability issues

Description

`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`

Obsolete Function	Standards	Risk	Replacement Function
`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.		

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`

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.

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
CERT C ID: MSC24-C, MSC33-C, POS33-C, PRE09-C

See Also

`Use of dangerous standard function` | `Unsafe standard function` | `Invalid use of standard library string routine`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Use of path manipulation function without maximum sized bu er checking

Destination buffer of `getwd` or `realpath` is smaller than `PATH_MAX` bytes

Description

Use of path manipulation function without maximum-sized bu er 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.

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.

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
ISO/IEC TS 17961 ID: libptr

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Use of plain char type for numerical value

Plain `char` variable in arithmetic operation without explicit signedness

Description

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.

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`.

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
CERT C ID: INT07-C

Introduced in R2016b

Use of previously closed resource

Function operates on a previously closed stream

Description

`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.

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.

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, 910
CERT C ID: FIO46-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`MISRA C:2012 Rule 22.6`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Use of previously freed pointer

Memory accessed after deallocation

Description

`Use of previously freed pointer` occurs when a block of memory is accessed after it is freed using the `free` function.

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.

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
CERT C ID: MEM00-C, MEM30-C
ISO/IEC TS 17961 ID: accfree, dblfree

See Also

Polyspace Analysis Options
Find defects (-checkers)

Polyspace Results
Deallocation of previously deallocated pointer

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Use of setjmp/longjmp

`setjmp` and `longjmp` cause deviation from normal control flow

Description

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.

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.

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
CERT C ID: MSC22-C

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

External Websites

Linux man page for setjmp

Introduced in R2015b

Use of tainted pointer

Pointer from an unsecure source may be NULL or point to unknown memory

Description

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 o set, 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

If you expect a valid memory location, check that the pointer is not NULL. Also, check the size of the memory location. This second check validates whether the size of the data the pointer points to matches the size your program expects.

Examples

Function to Change 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.

One 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;
    }
}
```

Result Information

Group: Tainted Data
Language: C | C++
Default: Off
Command-Line Syntax: TAINTED_PTR
Impact: Low
CWE ID: 690, 822
CERT C ID: API00-C, API02-C, ARR30-C, ARR38-C, EXP34-C, MEM10-C, MSC15-C
ISO/IEC TS 17961 ID: invptr, nullref

See Also

Pointer dereference with tainted offset

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Variable length array with nonpositive size

Size of variable-length array is zero or negative

Description

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.

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`.

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
CERT C ID: MEM04-C, MEM05-C

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Variable shadowing

Variable hides another variable of same name with nested scope

Description

`Variable shadowing` occurs when a variable hides another variable of the same name with nested scope.

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`.

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
CERT C ID: DCL01-C

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Vulnerable path manipulation

Path argument with `/../`, `/abs/path/`, or other unsecure elements

Description

`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.

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.

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
CERT C ID: FIO02-C

See Also

Use of path manipulation function without maximum sized buffer checking

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Vulnerable permission assignments

Argument gives read/write/search permissions to external users

Description

Vulnerable permission assignments looks at functions that can change file permissions, such as `chmod`, `umask`, `creat`, or `open`. If the specified permissions allow unintended actors to modify or read the resource, Bug Finder flags the functions as a defect.

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`.

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

CERT C ID: FIO06-C

See Also

Umask used with chmod-style arguments

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Vulnerable pseudo-random number generator

Using a cryptographically weak pseudo-random number generator

Description

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).

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.

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
CERT C ID: MSC30-C

See Also

`Deterministic random output from constant seed` | `Predictable random output from predictable seed` | `Unsafe standard encryption function`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Weak cipher algorithm

Encryption algorithm associated with the cipher context is weak

Description

`Weak cipher algorithm` 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.

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

Introduced in R2017a

Weak cipher mode

Encryption mode associated with the cipher context is weak

Description

`Weak cipher mode` 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.

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

Introduced in R2017a

Weak padding for RSA algorithm

Context used in encryption or signing operation is associated with insecure padding type

Description

`Weak padding for RSA algorithm` 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.

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

Incompatible padding for RSA algorithm operation | Missing blinding for RSA algorithm | Missing padding for RSA algorithm | Nonsecure RSA public exponent

Introduced in R2018a

Write without a further read

Variable never read after assignment

Description

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.

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.

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
CERT C ID: DCL22-C, MSC13-C

See Also

Polyspace Analysis Options
`Find defects (-checkers)`

Polyspace Results
`MISRA C:2012 Rule 2.2`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Writing to const quali ed object

Object declared with a `const` qualifier is modified

Description

`Writing to const quali ed 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:

Situation	Risk	Fix
You pass a `const`-qualified object as first argument of one of the following functions: • `mkstemp` • `mkostemp` • `mkostemps` • `mkdtemp`	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.	Pass a non-`const` object as first argument of the function.
You pass a `const`-qualified object as the destination argument of one of the following functions: • `strcpy` • `strncpy` • `strcat` • `memset`	These functions modify their destination argument. Therefore, they expect a modifiable `char` array as their destination argument.	Pass a non-`const` object as destination argument of the function.

Situation	Risk	Fix
You perform a write operation on a `const`-qualified 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.	Perform the write operation on a non-`const` object.

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.

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
CERT C ID: EXP40-C, MSC15-C, STR05-C, STR06-C, STR30-C
ISO/IEC TS 17961 ID: `strmod`

See Also

`Find defects (-checkers)`

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2015b

Writing to read-only resource

File initially opened as read only is modified

Description

`Writing to read-only resource` 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.

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 Comments”

`Introduced in R2015b`

Wrong allocated object size for cast

Allocated memory does not match destination pointer

Description

`Wrong allocated object size for cast` occurs during pointer conversion when the pointer's address is unaligned. 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.

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*` in line 5. 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.

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.

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.

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
CERT C ID: EXP36-C, MEM02-C, STR38-C

ISO/IEC TS 17961 ID: ALIGNCONV, INSUFMEM

See Also

Polyspace Analysis Options
Find defects (-checkers)

Polyspace Results
Unreliable cast of pointer

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

Wrong type used in sizeof

`sizeof` argument does not match pointed type

Description

`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`*`))`.

Rationale

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.

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
CERT C ID: ARR00-C, ARR01-C, MEM02-C, MEM35-C
ISO/IEC TS 17961 ID: insufmem

See Also

Find defects (-checkers)

Topics

“Interpret Polyspace Bug Finder Results”
“Address Polyspace Results Through Bug Fixes or Comments”

Introduced in R2013b

4

Functions, Properties, Classes, and Apps

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.

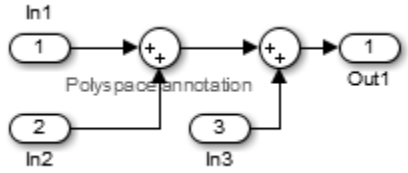
In the generated code associated with the annotated block, Polyspace adds code comments before and after the lines of code. Polyspace reads these comments and marks Polyspace results of the specified `kind` with the annotated information.

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.

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”.

`type` Value	`kind` Values
`'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++ Coding 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(matlabroot, '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 single 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.

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(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(target,opts)` analyzes `target` with the configuration options from the options object `opts`. It returns the location of the results folder.

`[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`.

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,'ModelReferenceRTWTargetOnly');

% 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.

```
% Copy demo files into a temporary folder
coderdemo_setup('coderdemo_averaging_filter');

% Generate code
codeName = 'averaging_filter';
codegenFolder = fullfile(pwd, 'codegenFolder');
codegen(codeName, '-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
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`.

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 9-18.

`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 9-18.

See Also

`pslinkfun` | `pslinkoptions` | pslinkoptions

Topics

“Analyze S-Function Code”
“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.

Alternatively, you can use the function `polyspaceBugFinderNoDesktop` with the syntax `polyspaceBugFinderNoDesktop(projectfile)`.

`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.

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 *`matlabroot`*`\polyspace\examples\cxx\Bug_Finder_Example`.

Open the project `Bug_Finder_Example.psprj` in the Polyspace interface.

```
prjFile = fullfile(matlabroot, '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 *`matlabroot`*`\polyspace\examples\cxx\Bug_Finder_Example\Results`.

Open the results of `resFolder`.

```
resFolder = fullfile(matlabroot, 'polyspace', 'examples',  ...
         'cxx', 'Bug_Finder_Example', 'Results');
polyspaceBugFinder('-results-dir',resFolder)
```

Run Polyspace Analysis with Options Object

This example shows how to run a Polyspace analysis from the MATLAB command-line. For this example:

- Save a C source file, `source.c`, in the folder `C:\Polyspace_Sources`.
- Save an include file in the folder `C:\Polyspace_Includes`.

Create an options object and add the source file and include folder to the properties.

```
opts = polyspace.BugFinderOptions;
opts.Sources = {'C:\Polyspace_Sources\source.c'};
opts.EnvironmentSettings.IncludeFolders = {'C:\Polyspace_Includes'};
opts.ResultsDir = 'C:\Polyspace_Results';
```

Polyspace runs on the file `C:\Polyspace_Sources\source.c` and stores the result in `C:\Polyspace_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. For this example:

- Save a C source file, `source.c`, in the folder `C:\Polyspace_Sources`.
- Save an include file in the folder `C:\Polyspace_Includes`.

To analyze `C:\Polyspace_Sources\source.c`, run the following command.

```
polyspaceBugFinder('-sources','C:\Polyspace_Sources\source.c', ...
    '-I','C:\Polyspace_Includes', ...
    '-results-dir','C:\Polyspace_Results')
```

To view the results, enter:

```
polyspaceBugFinder('-results-dir','C:\')
```

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(matlabroot, '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);
```

- “Run Polyspace Analysis by Using MATLAB Scripts”

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 single 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.

`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.

Examples

Create Polyspace Project from Make File

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.

```
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.

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 pattern;
```

`pattern` is a glob pattern that corresponds to folders or files you filter in or out of your project. 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 the command `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 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
`-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 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` throws an error if a project with the same name already exists in the output folder. Use this option to overwrite the project.
`-author`	Author name	Name of project author. Example: `-author jsmith`
`-debug`	None	Option used by MathWorks technical support
`-help`	None	Option to display the full list of `polyspace-configure` commands

Option	Argument	Description

`-lang`	`auto` (default)
	`c`
	`cpp`
	`cpp11`
		Option to specify source code language. The languages are:
		• C99: Use argument `c`.
		• C++03: Use argument `cpp`.
		• C++11: Use argument `cpp11`.
		By default,`polyspace-configure` detects the language.
		The argument to this option maps to the `Source code language` option in your Polyspace project. See `Source code language (-lang)`.

`-output-options-file`	None	Option to create a Polyspace analysis options file. Use this file for command-line analysis using `polyspace-bug-finder-nodesktop`.
`-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/`.
`-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`.

Option	Argument	Description
`-silent` (default) `-verbose`	None	Option to suppress or display additional messages from running `polyspace-configure`.

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 *`matlabroot`*`\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` 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` with the `-no-project` option. If you use this option, you do not need to specify the `buildCommand` argument.

Option	Argument	Description
`-extra-project-options`	Options to use for subsequent Polyspace analysis. For instance, `"-ignore-constant-overflows"`.	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 `"-ignore-constant-overflows -stubbed-pointers-are-unsafe"`. Suppose you have to set the option `-ignore-constant-overflows` 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 "-ignore-constant-overflows"` 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.

Cache Control Options

Option	Argument	Description

`-no-cache`
`-cache-sources` (default)
`-cache-all-files`
	None	Option to perform one of the following:
		• Not create a cache
		• Cache only source and header files.
		• Cache all files including binaries.

`-keep-cache` `-no-keep-cache` (default)	None	Option to preserve or clean up cache information after `polyspace-configure` completes execution. If `polyspace-configure` fails, you can provide this cache information to technical support for debugging purposes.
`-cache-path`	Path	Location of folder where cache information is stored. Example: `-cache-path ../cache`
`-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` 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` 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 troublehsoot 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`.

See Also

Topics

"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 Distributed Computing 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 2-36 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 MJS scheduler to run Polyspace remotely and monitor your jobs through the queue.

Before performing this example, set up an MJS 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.

```
mkdir 'C:\psdemo\src'
demo = fullfile(matlabroot,'polyspace','examples','cxx',...
'Bug_Finder_Example','sources');
copyfile(demo,'C:\psdemo\src\')
```

Submit two jobs to your scheduler.

```
polyspaceBugFinder -batch -scheduler myMJS@myCompany.com
    -sources C:\psdemo\src\*.c'
    -results-dir 'C:\psdemo\res1'
polyspaceBugFinder -batch -scheduler myMJS@myCompany.com
    -sources 'C:\psdemo\src\numeric.c'
    -results-dir 'C:\psdemo\res2'
```

```
    -add-to-results-repository
polyspaceJobsManager('listjobs','-scheduler','myMJS@myCompany.com')
```

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.

```
polyspaceJobsManager('download','-job','20','-results-folder', ...
   'C:\psdemo\res3','-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 Distributed Computing Server cluster (*`NodeHost`*).
- Name of the MJS 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)
“Run Polyspace Analysis on Remote Clusters Using Scripts” (Polyspace Code Prover)

Introduced in R2013b

polyspace-configure

Create Polyspace project from your build system at the DOS or UNIX command line

Syntax

```
polyspace-configure buildCommand

polyspace-configure buildCommand [OPTIONS]
```

Description

`polyspace-configure buildCommand` traces your build system and creates a Polyspace project with information gathered from your build system.

`polyspace-configure buildCommand [OPTIONS]` 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 Make file

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 Di erent 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 pattern
```

`pattern` is a glob pattern that corresponds to folders or files you filter in or out of your project. 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 Make File

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-nodesktop -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
`-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 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` throws an error if a project with the same name already exists in the output folder. Use this option to overwrite the project.
`-author`	Author name	Name of project author. `Example: -author jsmith`
`-debug`	None	Option used by MathWorks technical support
`-help`	None	Option to display the full list of `polyspace-configure` commands

Option	Argument	Description

`-lang`	`auto` (default)
	
	`c`
	
	`cpp`
	
	`cpp11`
		Option to specify source code language. The languages are:
		• C99: Use argument `c`.
		• C++03: Use argument `cpp`.
		• C++11: Use argument `cpp11`.
		
		By default,`polyspace-configure` detects the language.
		
		The argument to this option maps to the `Source code language` option in your Polyspace project. See `Source code language (-lang)`.

`-output-options-file`	None	Option to create a Polyspace analysis options file. Use this file for command-line analysis using `polyspace-bug-finder-nodesktop`.
`-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/`.
`-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`.

Option	Argument	Description
`-silent` (default) `-verbose`	None	Option to suppress or display additional messages from running `polyspace-configure`.

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 *`matlabroot`*`\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` 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` with the `-no-project` option. If you use this option, you do not need to specify the `buildCommand` argument.

Option	Argument	Description
`-extra-project-options`	Options to use for subsequent Polyspace analysis. For instance, `"-ignore-constant-overflows"`.	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 `"-ignore-constant-overflows -stubbed-pointers-are-unsafe"`. Suppose you have to set the option `-ignore-constant-overflows` 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 "-ignore-constant-overflows"` 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.

Cache Control Options

Option	Argument	Description

`-no-cache`
`-cache-sources` (default)
`-cache-all-files`
	None	Option to perform one of the following:
		• Not create a cache
		• Cache only source and header files.
		• Cache all files including binaries.

`-keep-cache` `-no-keep-cache` (default)	None	Option to preserve or clean up cache information after `polyspace-configure` completes execution. If `polyspace-configure` fails, you can provide this cache information to technical support for debugging purposes.
`-cache-path`	Path	Location of folder where cache information is stored. Example: `-cache-path ../cache`
`-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` 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` 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 troublehsoot 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`.

Introduced in R2013b

polyspace-bug- nder-nodesktop

Run a Bug Finder analysis from the DOS or UNIX command line

Syntax

```
polyspace-bug-finder-nodesktop
polyspace-bug-finder-nodesktop -sources sourceFiles [OPTIONS]

polyspace-bug-finder-nodesktop -sources-list-file listOfSources
[OPTIONS]

polyspace-bug-finder-nodesktop -options-file optFile

polyspace-bug-finder-nodesktop -h[elp]
```

Description

`polyspace-bug-finder-nodesktop [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-nodesktop -sources sourceFiles [OPTIONS]` runs a Bug Finder analysis on the source file or files `sourceFiles`. You can customize the analysis with additional options.

`polyspace-bug-finder-nodesktop -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-nodesktop -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-nodesktop -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 the Polyspace Bug Finder example. To run this example, replace *`matlabroot`* with the path to your MATLAB installation, for example `C:\Program Files\MATLAB\R2017a`.

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.

```
matlabroot\polyspace\bin\polyspace-bug-finder-nodesktop^
 -sources ^
matlabroot\polyspace\examples\cxx\Bug_Finder_Example\sources\numerical.c,^
matlabroot\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.

```
matlabroot\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 in the command itself. This example uses source files from the Polyspace Bug Finder example. To run this example, replace *`matlabroot`* with the path to your MATLAB installation, for example `C:\Program Files\MATLAB\R2017a`.

Save this text to a text file called `myOptionsFile.txt`.

```
# Options for analyzing numerical.c and programming.c
-sources matlabroot\polyspace\examples\cxx\Bug_Finder_Example\sources\numerical.c
-sources matlabroot\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.

```
matlabroot\polyspace\bin\polyspace-bug-finder-nodesktop -options-file myOptionsFile.txt
```

Open the results.

```
matlabroot\polyspace\bin\polyspace C:\Polyspace_Workspace\Results\^
ps_results.psbf
```

To rerun the analysis, you must rerun it from the command line.

- “Run Polyspace Analysis from Command Line”
- “Run Polyspace Analysis on Remote Clusters Using Scripts”

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. 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. 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”
“Run Polyspace Analysis on Remote Clusters Using Scripts”
“Analysis Options”

Introduced in R2013b

polyspace-results-repository

Upload, download and otherwise interact with results in the Polyspace Metrics repository

Syntax

```
polyspace-results-repository -upload resultsFolder -product
productName -prog projectName -verif-version runNumber [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 runNumber [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=${module_name//\//.}
    polyspace-results-repository          \
        -f                                \
        -server localhost                 \
        -upload "O$module"O               \
        -prog APPLICATION_NAME            \
        -module $module_name              \
        -verif-version "$RESULTS_VERSION"O
done
```

Input Arguments

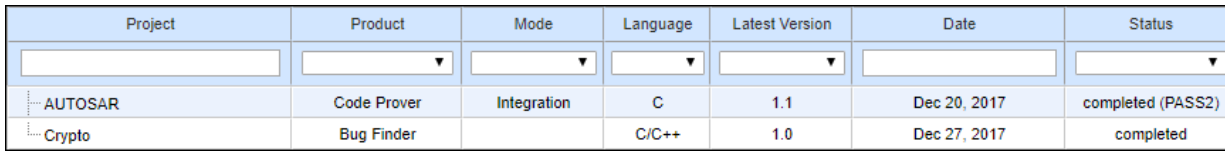
`resultsFolder` Folder containing Polyspace results
string

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\prover"`

`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"`

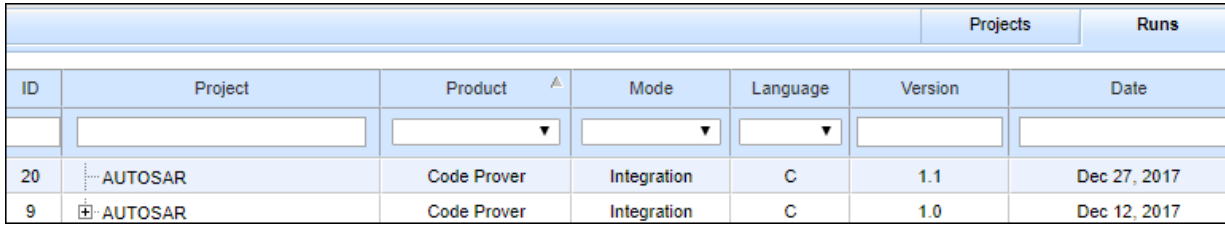
`newProjectName` Name of Polyspace project
string

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.

Option	Description
`-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.
`-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 Bug Finder

Identify software defects via static analysis

Description

The Polyspace Bug Finder app uses static analysis to quickly find run-time errors, data flow problems, and other defects in C and C++ code.

You can also add check compliance with MISRA C, MISRA C++, JSF++, and custom coding rules.

Open the Polyspace Bug Finder App

- MATLAB Toolstrip: On the Apps tab, under Code Verification, click the app icon.
- MATLAB command prompt: Enter `polyspaceBugFinder`.

Examples

- "Find Defects from the Polyspace Environment"
- "Run Polyspace Analysis from Command Line"

Programmatic Use

```
polyspaceBugFinder
```

See Also

Apps
Polyspace Code Prover

Functions

`polyspaceBugFinder` | `polyspaceConfigure`

Topics

“Find Defects from the Polyspace Environment”
“Run Polyspace Analysis from Command Line”
“Polyspace Bug Finder”

Introduced in R2013b

polyspace.Project class

`Package`: polyspace

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.

Construction

`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.Options.

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 (`.psprj` file).

  ```
  proj = polyspace.Project;
  projectLocation = fullfile(matlabroot, '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. Before obtaining the options, generate code from the model.

  ```
  modelName = 'sldemo_bounce';
  load_system(modelName);

  % Set parameters for Embedded Coder target
  set_param(modelName, 'SystemTargetFile', 'ert.tlc');
  set_param('sldemo_bounce','Solver','FixedStepAuto');
  set_param('sldemo_bounce','SupportContinuousTime','on')


  % Generate code
  rtwbuild(modelName);

  % 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(matlabroot, 'polyspace', 'examples',...
      'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
  proj.Configuration.ResultsDir = fullfile(pwd,'results');

  proj.run('bugFinder');

  resTable = proj.Results.getSummary('defects');
  ```

  For more information, see `polyspace.BugFinderResults.getSummary` or `polyspace.CodeProverResults.getSummary`.
- `getResults`: Obtain the full results or a more readable format into a MATLAB table.

  ```
  proj = polyspace.Project;
  proj.Configuration.Sources = {fullfile(matlabroot, 'polyspace', 'examples',...
      'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
  proj.Configuration.ResultsDir = fullfile(pwd,'results');

  proj.run('bugFinder');

  resTable = proj.Results.getResults('readable');
  ```

  For more information, see `polyspace.BugFinderResults.getResults` or `polyspace.CodeProverResults.getResults`.

Methods

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(matlabroot, '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');
```

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(matlabroot, '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 = proj.run('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(matlabroot, '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 = proj.run('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, use `polyspace.ModelLinkOptions` instead.

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.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 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(matlabroot, '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);
```

Open the results in the Polyspace user interface.

```
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(matlabroot, '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);
```

- “Run Polyspace Analysis by Using MATLAB Scripts”
- “Generate MATLAB Scripts from Polyspace User Interface”

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.

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 = 'sldemo_bounce';
load_system(modelName);
```

```
% Set parameters for Embedded Coder target
set_param(modelName, 'SystemTargetFile', 'ert.tlc');
set_param('sldemo_bounce','Solver','FixedStepAuto');
set_param('sldemo_bounce','SupportContinuousTime','on')


if exist(fullfile(pwd,'sldemo_bounce_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');
```

- “Run Polyspace Analysis by Using MATLAB Scripts”

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.

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(matlabroot, '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(matlabroot, '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);
```

- “Run Polyspace Analysis by Using MATLAB Scripts”

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.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.

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);
```

- “Run Polyspace Analysis by Using MATLAB Scripts”

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.DefectsOptions class

`Package`: polyspace

Create custom list of defects to check

Description

Create a custom list of defects to check. This object is useful if you want to check only a custom subset of the Bug Finder defects. To use your custom list of defects in an analysis, you must add it to a `polyspace.BugFinderOptions` or `polyspace.ModelLinkBugFinderOptions` object. In your Bug Finder options object, set the following properties:

- Add your defect options object to the `BugFinderAnalysis.CheckersList` property.
- Change the `BugFinderAnalysis.CheckersPreset` property to `'custom'`.

Construction

`defectList = polyspace.DefectsOptions` creates the defect options object `defectList`. 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, found on the individual defect reference pages.

By default, all defects are off. To turn on a defect, set the defect to true. For example:

```
defectList.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

Use a polyspace.DefectsOptions object to customize the list of defects checked during a Polyspace Bug Finder analysis.

Create options objects.

```
defects = polyspace.DefectsOptions;
opts = polyspace.Options;
```

Set Bug Finder object properties to analyze with the customized defect list.

```
opts.BugFinderAnalysis.CheckersList = defects;
opts.BugFinderAnalysis.CheckersPreset = 'custom';
```

Activate the numerical defects.

```
defects.FLOAT_ZERO_DIV = true;
defects.INT_ZERO_DIV = true;
defects.FLOAT_ABSORPTION = true;
defects.BITWISE_NEG = true;
defects.FLOAT_CONV_OVFL = true;
defects.FLOAT_OVFL = true;
defects.INT_CONV_OVFL = true;
defects.INT_OVFL = true;
defects.FLOAT_STD_LIB = true;
defects.INT_STD_LIB = true;
defects.SHIFT_NEG = true;
defects.SHIFT_OVFL = true;
defects.SIGN_CHANGE = true;
defects.UINT_CONV_OVFL = true;
defects.UINT_OVFL = true;
defects.BAD_PLAIN_CHAR_USE = true;
```

See Also

`polyspace.BugFinderOptions` | `polyspace.CodingRulesOptions` | `polyspace.ModelLinkBugFinderOptions`

Topics

"Defects"

Introduced in R2016b

polyspace.GenericTargetOptions class

`Package`: polyspace

Create a generic target configuration

Description

If your target processor does not match one of the preset targets on page 1-12, use this object to create a custom generic target. To use your custom target in an analysis, you must add it to a `polyspace.BugFinderOptions` or `polyspace.ModelLinkBugFinderOptions` object. In your options object, add your generic target options object to the `TargetCompiler.Target` property.

Construction

`genericTarget = polyspace.GenericTargetOptions` creates a generic target that you can customize. To specify the size and alignment of types, change the properties of the `genericTarget` object.

Properties

For more details about any of these properties, 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` De ne 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 polyspace.GenericTargetOptions object to customize a generic target for your analysis.

Create options objects.

```
target = polyspace.GenericTargetOptions;
opts = polyspace.Options;
```

Add the custom target to the Bug Finder options object.

```
opts.TargetCompiler.Target = target;
```

Customize the generic target.

```
target.Endianess = 'big';
target.LongLongNumBits = 64;
target.ShortNumBits = 8;
```

See Also

`Target processor type (-target)` | `polyspace.BugFinderOptions` | `polyspace.ModelLinkBugFinderOptions`

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 for one of the supported standard coding rule sets. To use your custom target in an analysis, you must add it to a `polyspace.Options` or `polyspace.ModelLinkOptions` object. In your options object:

- Add your coding rules options object to one of the `CodingRulesCodeMetrics.`*`RULESET`*`Subset` properties.
- Activate your coding rule set with one of the `CodingRulesCodeMetrics.Enable`*`RULESET`* properties.

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`

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 for that rule set. By default, all rules are on. To turn off a rule, set the rule to false. For example:

```
ruleList.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 during your analysis by using a coding rules options object.

Create options objects.

```
misraRules = polyspace.CodingRulesOptions('misraC2012');
opts = polyspace.Options;
```

Add the customized list of coding rules to the Bug Finder options object and activate them.

```
opts.CodingRulesCodeMetrics.MisraC3Subset = misraRules;
opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
```

Customize the coding rule list by turning off rules 2.1-2.7.

```
misraRules.rule_2_1 = false;
misraRules.rule_2_2 = false;
misraRules.rule_2_3 = false;
misraRules.rule_2_4 = false;
misraRules.rule_2_5 = false;
```

```
misraRules.rule_2_6 = false;
misraRules.rule_2_7 = false;
```

See Also

`polyspace.BugFinderOptions` | `polyspace.ModelLinkBugFinderOptions`

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.

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(matlabroot,'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(matlabroot, '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');
```

Alternatives

To read Code Prover results from MATLAB, use the class `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 `.prprj` 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 nishes
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 a ects 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 in nite loop
1000 (default) | positive integer

This property a ects 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` | `CERT-rules` | `CERT-all` | `ISO-17961` | `CWE` | `custom`

This property a ects Bug Finder analysis only.

Preset checker list, specified as a character vector of one of the preset options: `default`, `all`, or `custom`. To use `custom`, specify a `BugFinderAnalysis.CheckersList`.

For more information, see `Find defects (-checkers)`.

Example: `opts.BugFinderAnalysis.CheckersPreset = 'all'`

`EnableCheckers` Activate defect checking
true (default) | false

This property a ects 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`

ChecksAssumption (A ects Code Prover Only)

`AllowNegativeOperandInShift` Allow left shift operations on a negative number
false (default) | true

This property a ects 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 in nities 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 oating point values
`allow` (default) | `warn-first` | `warn-all` | `forbid`

This property a ects 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 a ects 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 a ects 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`

`IgnoreConstantOverflows` Allow overflow in computations involving constants
false (default) | true

This property affects Code Prover analysis only.

Allow overflow in computations involving constants, specified as true or false.

For more information, see `Ignore overflowing computations on constants (-ignore-constant-overflows)`.

Example: `opts.ChecksAssumption.IgnoreConstantOverflows = 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`

`ScalarOverflowsBehavior` Behavior of scalar overflows
`truncate-on-error` (default) | `wrap-around`

This property affects Code Prover analysis only.

Behavior of scalar overflows, specified as `wrap-around` or `truncate-on-error`.

For more information, see `Overflow computation mode (-scalar-overflows-behavior)`.

Example: `opts.ChecksAssumption.ScalarOverflowsBehavior = 'truncate-on-error'`

`ScalarOverflowsChecks` Check for integer overflows on signed and unsigned variables
`signed` (default) | `signed-and-unsigned` | `none`

This property affects Code Prover analysis only.

Check for integer overflows on signed and unsigned variables, specified as `signed`, `signed-and-unsigned`, or `none`.

For more information, see `Detect overflows (-scalar-overflows-checks)`.

Example: `opts.ChecksAssumption.ScalarOverflowsChecks = 'signed-and-unsigned'`

`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'`

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 `custom=class1[,class2,...]`.

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 `custom=method1[,method2,...]`.

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 a ects 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 les
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 list of comma-separated 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 list of comma-separated 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 le independently of other source les
false (default) | true

This property a ects 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 le during a le-by- le veri cation
cell array of file paths

This property a ects 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 | 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 MISRA AC AGC custom coding rules object. To create a custom coding rules object, see `polyspace.CodingRulesOptions`.
- Full path to a file containing your MISRA AC AGC subset. You can create this file manually or in the Polyspace interface. See “Check for Coding Rule Violations”.

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`

`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`

`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)`.

Example: `opts.CodingRulesCodeMetrics.CodeMetrics = true`

`CustomRulesSubset` Custom naming conventions to check against
custom coding rules file

Custom naming conventions to check against, specified as a custom coding rules file. You can create the custom coding rules file manually or in the Polyspace interface.

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.CustomRulesSubset = 'C:\ps_settings\coding_rules\custom_rules.txt'`

Data Types: `char`

`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;`

`EnableCustomRules` Check custom coding rules
false (default) | true

Check custom coding rules, specified as true or false. Use with `CustomRulesSubset`.

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCustomRules = 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;`

`JsfSubset` `Subset of JSF C++ rules to check`
`shall-rules` (default) | `shall-will-rules` | `all-rules` | `polyspace.CodingRulesOptions` object | 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 JSF C++ custom coding rules object. To create a custom coding rules object, see `polyspace.CodingRulesOptions`.
- Full path to a file containing your JSF C++ subset. You can create this file manually or from the Polyspace interface. See “Check for Coding Rule Violations”.

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` | `CERT-rules` | `CERT-all` | `ISO-17961` | `all` | `SQO-subset1` | `SQO-subset2` | `polyspace.CodingRulesOptions` object | 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 MISRA C:2012 custom coding rules object. To create a custom coding rules object, see `polyspace.CodingRulesOptions`.
- Full path to a file containing your MISRA C:2012 subset. You can create the custom coding rules file manually or in the Polyspace interface. See “Check for Coding Rule Violations”.

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) | `all-rules` | `SQO-subset1` | `SQO-subset2` | `single-unit-rules` | `system-decidable-rules` | `polyspace.CodingRulesOptions` object | 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 MISRA C:2004 custom coding rules object. To create a custom coding rules object, see `polyspace.CodingRulesOptions`.
- Full path to a file containing your MISRA C:2004 subset. You can create the custom coding rules file manually or in the Polyspace interface. See “Check for Coding Rule Violations”.

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 | 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 MISRA C++ coding rules object. To create a custom coding rules object, see `polyspace.CodingRulesOptions`.
- Full path to a file containing your MISRA C++ subset. You can create this file manually or from the Polyspace interface. See “Check for Coding Rule Violations”.

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=`*`file1`*`[,`*`folder1`*`[,...]]`

Files on which you do not want analysis results, specified by `include-folders`, `all-headers`, or a character array beginning with `custom=` and containing a list of comma-separated 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=`*`file1`*`[,`*`folder1`*`[,...]]`

Files on which you want analysis results, specified by `source-headers`, `all-headers`, or a character array beginning with `custom=` and containing a comma-separated 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 a ects Code Prover analysis only.

Functions to stub during analysis, specified as a cell array of function names.

For more information, see .

Example: `opts.InputsStubbing.FunctionsToStub = {'func1', 'func2'}`

`NoDefInitGlob` Consider global variables as uninitialized
false (default) | true

This property a ects Code Prover analysis only.

Consider global variables as uninitialized, specified as true or false.

For more information, see .

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 .

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 unde ne
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` | `SecurityCERT` | `SecurityCWE` | `SecurityISO_17961` | `CodeMetrics` | `CodingRules`

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` | `CodingRules` | `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 a ects 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 con guration from external le de nitions
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` Con gure 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 a ects 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 les to parse for multitasking con guration
`auto` (default) | `custom=`*`file1`*`[,`*`folder1`*`[,...]]`

Specify the path to the OIL files the software parses to set up your multitasking configuration:

- In `auto` mode, the analysis uses OIL files in your project source and include folders, but not their subfolders.
- In `custom` mode, 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 (A ects Code Prover Only)

`ContextSensitivity` Store call context information to identify function call that caused errors
none (default) | `auto` | `custom=`*`function1`*`[,`*`function2`*`[,...]]`

This property a ects 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 les you want to verify at higher precision
cell array of file names and precision levels

This property a ects 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 veri cation
2 (default) | 0 | 1 | 3

This property a ects 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 (A ects Code Prover Only)

`Inline` Functions on which separate results must be generated for each function call
cell array of function names

This property a ects 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 a ects 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` | `clang3.x` | `visual9.0` | `visual10` | `visual11.0` | `visual12.0` | `visual14.0` | `keil` | `iar` | `codewarrior` | `diab` | `greenhills` | `iar-ew` | `tasking` | `ti`

Compiler that builds your source code.

For more information, see `Compiler (-compiler)`.

Example: `opts.TargetCompiler.Compiler = 'Visual11.0'`

`Cpp11Extension` Allow C++11 language extensions
false (default) | true

Allow C++11 language extensions, specified as true or false.

For more information, see `C++11 extensions (-cpp11-extension)`.

Example: `opts.TargetCompiler.Cpp11Extension = true`

`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'`

`NoLanguageExtensions` Restrict analysis to C language specified in ANSI C standard
false (default) | true

Restrict analysis to the C language that is specified in the ANSI C standard, specified as true or false. For more information, see `Respect C90 standard (-no-language-extensions)`.

Example: `opts.TargetCompiler.NoLanguageExtensions = true`

`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) | `sparc` | `m68k` | `powerpc` | `powerpc64` | `c-167` | `tms320c3x` | `sharc21x61` | `necv850` | `hc08` | `hc12` | `mpc5xx` | `c18` | `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'`

Veri cationAssumption (A ects Code Prover Only)

`ConsiderVolatileQualifierOnFields` Assume that volatile quali ed structure elds 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 a ects 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 justi cations 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.

For more information, see `-import-comments`

Example: `opts.ImportComments = fullfile(matlabroot,'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`.

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` | `CERT-rules` | `CERT-all` | `ISO-17961` | `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`, or `custom`. To use `custom`, specify a `BugFinderAnalysis.CheckersList`.

For more information, see `Find defects (-checkers)`.

Example: `opts.BugFinderAnalysis.CheckersPreset = 'all'`

`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`

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`

`IgnoreConstantOverflows` Allow overflow in computations involving constants
false (default) | true

This property affects Code Prover analysis only.

Allow overflow in computations involving constants, specified as true or false.

For more information, see `Ignore overflowing computations on constants (-ignore-constant-overflows)`.

Example: `opts.ChecksAssumption.IgnoreConstantOverflows = 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`

`ScalarOverflowsBehavior` Behavior of scalar overflows
`truncate-on-error` (default) | `wrap-around`

This property affects Code Prover analysis only.

Behavior of scalar overflows, specified as `wrap-around` or `truncate-on-error`.

For more information, see `Overflow computation mode (-scalar-overflows-behavior)`.

Example: `opts.ChecksAssumption.ScalarOverflowsBehavior = 'truncate-on-error'`

`ScalarOverflowsChecks` Check for integer overflows on signed and unsigned variables
`signed` (default) | `signed-and-unsigned` | `none`

This property affects Code Prover analysis only.

Check for integer overflows on signed and unsigned variables, specified as `signed`, `signed-and-unsigned`, or `none`.

For more information, see `Detect overflows (-scalar-overflows-checks)`.

Example: `opts.ChecksAssumption.ScalarOverflowsChecks = 'signed-and-unsigned'`

`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'`

CodeProverVerification (Affects Code Prover only)

`ClassAnalyzer` Classes that you want to verify
none (default) | `all` | cell array of class names

This property affects Code Prover analysis only.

Classes that you want to verify, specified as `all`, `none`, or a cell array 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` | cell array of function names

This property affects Code Prover analysis only.

Functions that the generated main must call in the cyclic code loop, specified as `none`, `all`, or a cell array 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` | cell array of variable names

This property affects Code Prover analysis only.

Variables that the generated main must initialize before the cyclic code loop, specified as `none`, `all`, or a cell array 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` | cell array of variable names

This property affects Code Prover analysis only.

Variables that the generated main must initialize in the cyclic code loop, specified as `none`, `all`, or a cell array 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 | 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 MISRA AC AGC custom coding rules object. To create a custom coding rules object, see `polyspace.CodingRulesOptions`.
- Full path to a file containing your MISRA AC AGC subset. You can create this file manually or in the Polyspace interface. See “Check for Coding Rule Violations”.

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`

`BooleanTypes` Data types the coding rule checker must treat as e ectively 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`

`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)`.

Example: `opts.CodingRulesCodeMetrics.CodeMetrics = true`

`CustomRulesSubset` Custom naming conventions to check against
custom coding rules file

Custom naming conventions to check against, specified as a custom coding rules file. You can create the custom coding rules file manually or in the Polyspace interface.

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.CustomRulesSubset = 'C:\ps_settings\coding_rules\custom_rules.txt'`

Data Types: `char`

`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;`

`EnableCustomRules` Check custom coding rules
false (default) | true

Check custom coding rules, specified as true or false. Use with `CustomRulesSubset`.

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCustomRules = 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;`

`JsfSubset` Subset of JSF C++ rules to check
`shall-rules` (default) | `shall-will-rules` | `all-rules` | `polyspace.CodingRulesOptions` object | 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 JSF C++ custom coding rules object. To create a custom coding rules object, see `polyspace.CodingRulesOptions`.
- Full path to a file containing your JSF C++ subset. You can create this file manually or from the Polyspace interface. See “Check for Coding Rule Violations”.

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` | `CERT-rules` | `CERT-all` | `ISO-17961` | `all` | `SQO-subset1` | `SQO-subset2` | `polyspace.CodingRulesOptions` object | 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 MISRA C:2012 custom coding rules object. To create a custom coding rules object, see `polyspace.CodingRulesOptions`.
- Full path to a file containing your MISRA C:2012 subset. You can create the custom coding rules file manually or in the Polyspace interface. See “Check for Coding Rule Violations”.

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) | `all-rules` | `SQO-subset1` | `SQO-subset2` | `single-unit-rules` | `system-decidable-rules` | `polyspace.CodingRulesOptions` object | 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 MISRA C:2004 custom coding rules object. To create a custom coding rules object, see `polyspace.CodingRulesOptions`.
- Full path to a file containing your MISRA C:2004 subset. You can create the custom coding rules file manually or in the Polyspace interface. See “Check for Coding Rule Violations”.

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 | 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 MISRA C++ coding rules object. To create a custom coding rules object, see `polyspace.CodingRulesOptions`.

- Full path to a file containing your MISRA C++ subset. You can create this file manually or from the Polyspace interface. See “Check for Coding Rule Violations”.

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=`*`file1`*`[,`*`folder1`*`[,...]]`

Files on which you do not want analysis results, specified by `include-folders`, `all-headers`, or a character array beginning with `custom=` and containing a list of comma-separated 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=`*`file1`*`[,`*`folder1`*`[,...]]`

Files on which you want analysis results, specified by `source-headers`, `all-headers`, or a character array beginning with `custom=` and containing a comma-separated 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 .

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 .

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 .

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` | `SecurityCERT` | `SecurityCWE` | `SecurityISO_17961` | `CodeMetrics` | `CodingRules`

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` | `CodingRules` | `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 a ects 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=`*`file1`*`[,`*`folder1`*`[,...]]`

Specify the path to the OIL files the software parses to set up your multitasking configuration:

- In `auto` mode, the analysis uses OIL files in your project source and include folders, but not their subfolders.
- In `custom` mode, 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 (A ects Code Prover Only)

`ContextSensitivity` Store call context information to identify function call that caused errors
none (default) | `auto` | `custom=`*`function1`*`[,`*`function2`*`[,...]]`

This property a ects 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 les you want to verify at higher precision
cell array of file names and precision levels

This property a ects 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 veri cation
2 (default) | 0 | 1 | 3

This property a ects 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 veri cation 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` | `clang3.x` | `visual9.0` | `visual10` | `visual11.0` | `visual12.0` | `visual14.0` | `keil` | `iar` | `codewarrior` | `diab` | `greenhills` | `iar-ew` | `tasking` | `ti`

Compiler that builds your source code.

For more information, see `Compiler (-compiler)`.

Example: `opts.TargetCompiler.Compiler = 'Visual11.0'`

`Cpp11Extension` Allow C++11 language extensions
false (default) | true

Allow C++11 language extensions, specified as true or false.

For more information, see `C++11 extensions (-cpp11-extension)`.

Example: `opts.TargetCompiler.Cpp11Extension = true`

`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'`

`NoLanguageExtensions` Restrict analysis to C language specified in ANSI C standard
false (default) | true

Restrict analysis to the C language that is specified in the ANSI C standard, specified as true or false. For more information, see `Respect C90 standard (-no-language-extensions)`.

Example: `opts.TargetCompiler.NoLanguageExtensions = true`

`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) | `sparc` | `m68k` | `powerpc` | `powerpc64` | `c-167` | `tms320c3x` | `sharc21x61` | `necv850` | `hc08` | `hc12` | `mpc5xx` | `c18` | `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'`

Verification Assumption (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.

For more information, see `-import-comments`

Example: `opts.ImportComments = fullfile(matlabroot,'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`.

Example: `opts.ResultsDir = 'C:\project\myproject\results\'`

`Sources` Source les
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 = 'diab';
```

Create another object and use copyTo to copy over options from the previous object.

```
opts2 = polyspace.Options();
copyTo(opts1, opts2);
```

See Also

`polyspace.BugFinderOptions` | `polyspace.ModelLinkBugFinderOptions` | `polyspace.Options` | `polyspace.Options.generateProject`

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`.

Input Arguments

`opts` — Options object to convert into a `psprj` file
`polyspace.Options` or `polyspace.ModelLinkOptions` object

Option object to 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(matlabroot, '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

`polyspace.BugFinderOptions` | `polyspace.ModelLinkBugFinderOptions` | `polyspace.Options` | `polyspace.Options.copyTo`

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

`polyspace.BugFinderOptions` | `polyspace.ModelLinkBugFinderOptions` | `polyspace.Options` | `polyspace.Options.generateProject`

Topics

“Generate MATLAB Scripts from Polyspace User Interface”

Introduced in R2017b

run

Class: polyspace.Project
Package: polyspace

Run a Polyspace analysis

Syntax

```
proj.run(product)
```

Description

`status = proj.run(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(matlabroot, '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');
```

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. See “Defects”.
`'misraC'`	MISRA C:2004 rules. See “MISRA C:2004 and MISRA AC AGC Rules”.

Entry	Meaning
`'misraCAGC'`	MISRA C:2004 rules for generated code. See “MISRA C:2004 and MISRA AC AGC Rules”.
`'misraCPP'`	MISRA C++ rules. See “MISRA C++:2008 Rules”.
`'misraC2012'`	MISRA C:2012 rules. See “MISRA C:2012 Directives and Rules”.
`'jsf'`	JSF C++ rules. See “JSF C++ Rules”.
`'metrics'`	Code complexity metrics. See “Code Metrics”.
`'customRules'`	Custom rules enforcing naming conventions for identifiers. See “Custom Coding Rules”.

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(matlabroot,'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(matlabroot, '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

```
getResults(content)
```

Description

`resTable = 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
`'full'` (default) | `'readable'`

Amount of information to be included for each result. If you specify `'full'`, all information is included. See “Export Polyspace Analysis Results”. 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.

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(matlabroot,'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(matlabroot, '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 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 Specification

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.

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.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps
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.

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.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps
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 Rule Violations Not Displayed”.

Check Information

Group: The implementation
Category: Required
AGC Category: Required
Language: C90, C99
CERT C: EXP11-C, FLP30-C

See Also

Topics

“Check for Coding Rule 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 De nition

All source les 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 Speci cation

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.

Message in Report

All source files shall compile without any compilation errors.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Check Information

Group: Compilation and build

Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 1.1

Topics

“Check for Coding Rule 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 Specification

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.

Message in Report

Run-time failures shall be minimized.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Check Information

Group: Code design
Category: Required
AGC Category: Required
Language: C90, C99
ISO/IEC TS 17961 ID: `chrsgnext, diverr, intoflow, inverrno, invfmtstr, ioileave, libptr, nonnullcs, nullref, taintformatio, taintsink, taintstrcpy, usrfmt, xfilepos`

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`

Topics

“Check for Coding Rule Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Dir 4.3

Assembly language shall be encapsulated and isolated

Description

Directive De nition

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 Speci cation

Polyspace does not raise a warning on assembly language code encapsulated in the following:

- `asm` functions or `asm` pragmas
- Macros

Message in Report

Assembly language shall be encapsulated and isolated

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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.

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
Language: C90, C99

See Also

MISRA C:2012 Rule 1.2

Topics

“Check for Coding Rule 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`.

Message in Report

Identifiers in the same name space with overlapping visibility should be typographically unambiguous.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Examples

Typographically Ambiguous Identi ers

```
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: Code design
Category: Advisory
AGC Category: Readability
Language: C90, C99
CERT C: DCL02-C

See Also

Topics

“Check for Coding Rule 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 Specification

The rule checker flags use of basic data types in variable or function declarations and definitions. The rule enforces use of `typedefs` instead.

The rule checker does not flag the use of basic types in the `typedef` statements themselves.

Message in Report

Typedefs that indicate size and signedness should be used in place of the basic numerical types

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Examples

Direct Use of Basic Types in De nitions

```
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
Language: C90, C99
CERT C: INT00-C

See Also

Topics

“Check for Coding Rule 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 De nition

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 Speci cation

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.

Message in Report

If a function returns error information, then that error information shall be tested.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Check Information

Group: Code design
Category: Required
AGC Category: Required
Language: C90, C99
CERT C: ERR00-C, API04-C
ISO/IEC TS 17961 ID: `inverrno, liberr`

See Also

Topics

"Check for Coding Rule Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2017a

MISRA C:2012 Directive 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 De nition

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 Speci cation

If a pointer to a structure or union is not dereferenced in a file but the structure or union is defined in the file or a header file included in the file, the checker flags a coding rule violation.

Message in Report

If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Examples

Object Implementation Revealed

```
/* file.h */

#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct  {
  int a;
} myStruct;

#endif

/* file.c */
#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 */

#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct myStruct *ptrMyStruct;

#endif
```

```
/* file.c */
#include "file.h"

ptrMyStruct getObj(void);
void useObj(ptrMyStruct);

void func() {
  ptrMyStruct sPtr = getObj();
  useObj(sPtr);
}
```

```
/* file2.c */
#include "file.h"

struct myStruct {
  int a;
};
```

Check Information

Group: Code design
Category: Advisory
AGC Category: Advisory
Language: C90, C99
CERT C: DCL12-C

See Also

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Rule 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 De nition

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 Speci cation

Polyspace considers all function-like macro definitions.

Message in Report

A function should be used in preference to a function-like macro where they are interchangeable

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Check Information

Group: Code design
Category: Advisory
AGC Category: Advisory
Language: C90, C99
CERT C: PRE00-C

See Also

MISRA C:2012 Rule 13.2 | MISRA C:2012 Rule 20.7

Topics

“Check for Coding Rule 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 De nition

Precautions shall be taken in order to prevent the contents of a header le 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 Speci cation

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.

Message in Report

Precautions shall be taken in order to prevent the contents of a header file being included more than once.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99
CERT C: PRE06-C

See Also

Topics

"Check for Coding Rule 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 Specification

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`

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.

Message in Report

The validity of values passed to library functions shall be checked

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Check Information

Group: Code design
Category: Required
AGC Category: Required
Language: C90, C99
ISO/IEC TS 17961 ID: `chrsgnext, inverrno, invfmtstr, libptr, nonnullcs, taintformatio, taintsink, taintstrcpy, usrfmt, xfilepos`

See Also

`MISRA C:2012 Dir 4.1`

Topics

“Check for Coding Rule 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 Specification

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”.

Message in Report

Functions which are designed to provide operations on a resource should be called in an appropriate sequence.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99
CERT C: MEM30-C

See Also

MISRA C:2012 Rule 22.1 | MISRA C:2012 Rule 22.2 | MISRA C:2012 Rule 22.6

Topics

"Check for Coding Rule 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 De nition

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.

Message in Report

The validity of values received from external sources shall be checked.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99
CERT C: EXP34-C, INT04-C, INT10-C, STR02-C
ISO/IEC TS 17961 ID: `diverr`, `nullref`, `taintformatio`, `taintnoproto`, `taintsink`, `taintstrcpy`, `usrfmt`

See Also

Topics

“Check for Coding Rule Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2017a

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 Specification

The rule violation can come from multiple causes. 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.

Message in Report

- Too many nesting levels of #includes: N1. The limit is N0.

  Note: The rule checker considers a brace as an additional level. For instance, the `if` branch in this code is counted as two levels of nesting.

  ```
  if(flag) {
  }
  ```

  The metric `Number of Call Levels` counts this as one level of nesting.
- Integer constant is too large.

- ANSI C does not allow '#XX'.
- Text following preprocessing directive violates ANSI standard.
- Too many macro definitions: N1. The limit is N0.
- Array of zero size should not be used.
- Integer constant does not fit within long int.
- Integer constant does not fit within unsigned long int.
- Too many nesting levels for control flow: N1. The limit is N0.
- Assembly language should not be used.
- Too many enumeration constants: N1. The limit is N0.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Check Information

Group: Standard C Environment
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 1.2

Topics

“Check for Coding Rule 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 Specification

All the supported extensions lead to a violation of this MISRA rule.

Message in Report

- ANSI C90 forbids hexadecimal floating-point constants.
- ANSI C90 forbids universal character names.
- ANSI C90 forbids mixed declarations and code.
- ANSI C90/C99 forbids case ranges.
- ANSI C90/C99 forbids local label declaration.
- ANSI C90 forbids mixed declarations and code.
- ANSI C90/C99 forbids typeof operator.
- ANSI C90/C99 forbids casts to union.
- ANSI C90 forbids compound literals.
- ANSI C90/C99 forbids statements and declarations in expressions.

- ANSI C90 forbids __func__ predefined identifier.
- ANSI C90 forbids keyword '_Bool'.
- ANSI C90 forbids 'long long int' type.
- ANSI C90 forbids long long integer constants.
- ANSI C90 forbids 'long double' type.
- ANSI C90/C99 forbids 'short long int' type.
- ANSI C90 forbids _Pragma preprocessing operator.
- ANSI C90 does not allow macros with variable arguments list.
- ANSI C90 forbids designated initializer.

  Keyword 'inline' should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Check Information

Group: Standard C Environment
Category: Advisory
AGC Category: Advisory
Language: C90, C99
CERT C: MSC04-C

See Also

MISRA C:2012 Rule 1.1

Topics

“Check for Coding Rule 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.

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 Rule Violations Not Displayed”.

Check Information

Group: Standard C Environment
Category: Required
AGC Category: Required
Language: C90, C99
ISO/IEC TS 17961 ID: `accfree, accsig, argcomp, chrsgnext, dblfree, diverr, funcdecl, intoflow, intptrconv, invfmtstr, invptr, ioileave, libmod, libptr, nonnullcs, ptrcomp, ptrobj, resident, restrict, taintsink, uninitref, usrfmt, xfilepos, xfree`

See Also

MISRA C:2012 Dir 4.1

Topics

“Check for Coding Rule 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 Specification

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.

Message in Report

A project shall not contain unreachable code.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99
CERT C: MSC07-C, MSC12-C

See Also

MISRA C:2012 Rule 14.3 | MISRA C:2012 Rule 16.4

Topics

“Check for Coding Rule 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 Specification

Polyspace Bug Finder detects useless write operations during analysis.

Message in Report

There shall be no dead code.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99
CERT C: DCL22-C, MSC12-C

See Also

MISRA C:2012 Rule 17.7 | Write without a further read

Topics

“Check for Coding Rule 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 De nition

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.

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 Rule 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
Language: C90, C99

See Also

MISRA C:2012 Rule 2.4

Topics

“Check for Coding Rule 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.

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 Rule 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
Language: C90, C99

See Also

MISRA C:2012 Rule 2.3

Topics

"Check for Coding Rule 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.

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 Rule 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
Language: C90, C99

See Also

Topics

“Check for Coding Rule 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.

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 Rule 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
Language: C90, C99

See Also

Topics

“Check for Coding Rule 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.

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 Rule 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
Language: C90, C99

See Also

Unused parameter

Topics

“Check for Coding Rule Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

Introduced in R2015b

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 Specification

You cannot annotate this rule in the source code.

For information on annotations, see “Annotate Code and Hide Known or Acceptable Results”.

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 Rule 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
Language: C90, C99

CERT C: MSC04-C

See Also

Topics

“Check for Coding Rule 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 De nition

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.

Message in Report

Line-splicing shall not be used in // comments.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C99

See Also

Topics

“Check for Coding Rule 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 De nition

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.

Message in Report

Octal and hexadecimal escape sequences shall be terminated.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99

See Also

Topics

“Check for Coding Rule 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 Specification

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.

Message in Report

Trigraphs should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Check Information

Group: Character Sets and Lexical Conventions
Category: Advisory
AGC Category: Advisory
Language: C90, C99
CERT C: PRE07-C

See Also

Topics

“Check for Coding Rule 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`.

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 option, Respect C90 Standard on page 1-36.

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 Rule 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
Language: C90, C99
CERT C: DCL23-C, DCL40-C
ISO/IEC TS 17961 ID: `funcdecl`

See Also

`MISRA C:2012 Rule 5.2` | `MISRA C:2012 Rule 5.4` | `MISRA C:2012 Rule 5.5`

Topics

“Check for Coding Rule 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

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 option, Respect C90 Standard on page 1-36.

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 Rule 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 Identi ers 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
Language: C90, C99
CERT C: DCL23-C

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`

Topics

"Check for Coding Rule 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 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 option, Respect C90 Standard on page 1-36.

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 Rule 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
Language: C90, C99
CERT C: DCL01-C, DCL23-C

See Also

MISRA C:2012 Rule 5.2 | MISRA C:2012 Rule 5.8

Topics

"Check for Coding Rule 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 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 option, Respect C90 Standard on page 1-36.

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 Rule 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
Language: C90, C99
CERT C: DCL23-C

See Also

MISRA C:2012 Rule 5.1 | MISRA C:2012 Rule 5.2 | MISRA C:2012 Rule 5.5

Topics

“Check for Coding Rule 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 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 option, Respect C90 Standard on page 1-36.

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 Rule 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
Language: C90, C99
CERT C: DCL23-C

See Also

`MISRA C:2012 Rule 5.1` | `MISRA C:2012 Rule 5.2` | `MISRA C:2012 Rule 5.4`

Topics

“Check for Coding Rule 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.

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 Rule 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:
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 5.7

Topics

“Check for Coding Rule 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.

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 Rule Violations Not Displayed”.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 5.6

Topics

“Check for Coding Rule 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.

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 Rule Violations Not Displayed”.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 5.3

Topics

“Check for Coding Rule 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 Specification

This rule checker assumes that rule 5.8 is not violated.

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 Rule Violations Not Displayed”.

Check Information

Group: Identifiers
Category: Advisory
AGC Category: Readability
Language: C90, C99

See Also

MISRA C:2012 Rule 8.10

Topics

“Check for Coding Rule 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` 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.

Message in Report

Bit-fields shall only be declared with an appropriate type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Check Information

Group: Types
Category: Required
AGC Category: Required

Language: C90, C99
CERT C: INT12-C

See Also

Topics

“Check for Coding Rule 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 Specification

This rule does not apply to unnamed bit fields because their values cannot be accessed.

Message in Report

Single-bit named bit fields shall not be of a signed type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Check Information

Group: Types
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Topics

“Check for Coding Rule 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 De nition

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 Speci cation

If you use octal constants in a macro definition, the rule checker flags the issue even if the macro is not used.

Message in Report

Octal constants shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
`Language`: C90, C99
`CERT C`: DCL18-C

See Also

Topics

“Check for Coding Rule 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.

Message in Report

A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Check Information

Group: Literals and Constants
Category: Required
AGC Category: Readability
Language: C90, C99

See Also

Topics

“Check for Coding Rule 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.

Message in Report

The lowercase character “l” shall not be used in a literal suffix.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Check Information

Group: Literals and Constants
Category: Required
AGC Category: Readability
Language: C90, C99
CERT C: DCL16-C

See Also

Topics

“Check for Coding Rule 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.

Message in Report

A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Rule Violations Not Displayed".

Examples

Incorrect Assignment of String Literal

```
char *str1 = "AccountHolderName";
const char *str2 = "AccountHolderName";

void checkAccount1(char*);              /* Non-Compliant */
void checkAccount2(const char*);        /* Compliant */

void main() {
 checkAccount1("AccountHolderName");    /* Non-Compliant */
 checkAccount2("AccountHolderName");    /* 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
Language: C90, C99
ISO/IEC TS 17961 ID: `strmod`

See Also

`MISRA C:2012 Rule 11.4` | `MISRA C:2012 Rule 11.8`

Topics

“Check for Coding Rule 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 De nition

Types shall be explicitly speci ed.

Rationale

n 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 Speci cation

The rule checker flags situations where a function parameter or return type is not explicitly specified. To enable checking of this rule, use the option `Respect C90 standard (-no-language-extensions)`.

Message in Report

Types shall be explicitly specified.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90
CERT C: DCL31-C

See Also

MISRA C:2012 Rule 8.2

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Rule 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 Specification

The rule checker shows a violation if the parameters in a function declaration or definition are missing names or data types.

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 Rule 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
Language: C90, C99
CERT C: DCL07-C, DCL20-C, DCL36-C
ISO/IEC TS 17961 ID: `argcomp, taintnoproto`

See Also

`MISRA C:2012 Rule 8.1` | `MISRA C:2012 Rule 8.4` | `MISRA C:2012 Rule 17.3`

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Rule 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 Specification

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.

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 Rule 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
Language: C90, C99
CERT C: DCL40-C, EXP37-C
ISO/IEC TS 17961 ID: `argcomp, funcdecl`

See Also

`MISRA C:2012 Rule 8.4`

Topics

"Avoid Violations of MISRA C 2012 Rules 8.x"
"Check for Coding Rule 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 Specification

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.

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 Rule Violations Not Displayed”.

Examples

De nition 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
Language: C90, C99
CERT C: DCL36-C
ISO/IEC TS 17961 ID: `funcdecl`

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`

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Rule 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 Specification

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.

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 Rule 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
Language: C90, C99

See Also

MISRA C:2012 Rule 8.4

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Rule 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 Specification

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.

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 Rule 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
Language: C90, C99

See Also

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Rule 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 Specification

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.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

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 Rule 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
Language: C90, C99
CERT C: DCL15-C, DCL19-C

See Also

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Rule 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 Specification

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.

Message in Report

The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Examples

Linkage Con ict 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`.

One possible correction is to use `static` and `extern` consistently:

```
static int foo = 0;
static int foo;

extern int hhh;
extern int hhh;
```

Linkage Con ict Between Function Declaration and De nition

```
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
Language: C90, C99
CERT C: DCL15-C, DCL36-C

See Also

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Rule 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 De nition

An object should be de ned at block scope if its identi er 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 Speci cation

The rule checker flags `static` objects that are accessed in one function only but declared at file scope.

Message in Report

An object should be defined at block scope if its identifier only appears in a single function.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99
CERT C: DCL19-C

See Also

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Rule 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 Specification

The rule checker flags definitions that contain the `inline` specifier without an accompanying `static` specifier.

Message in Report

An inline function shall be declared with the static storage class.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C99

See Also

MISRA C:2012 Rule 5.9

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Rule 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 Specification

The rule checker flags arrays declared with the `extern` specifier if the declaration does not explicitly specify the array size.

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 Rule 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
Language: C90, C99
CERT C: ARR02-C

See Also

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Rule 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 De nition

Within an enumerator list, the value of an implicitly-speci ed 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 Speci cation

The rule checker flags an enumeration if it has an implicitly specified enumeration constant with the same value as another enumeration constant.

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 Rule Violations Not Displayed”.

Examples

Replication of Value in Implicitly Speci ed 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
Language: C90, C99
CERT C: INT09-C

See Also

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Rule 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 Specification

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.

Message in Report

A pointer should point to a const-qualified type whenever possible.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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.

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
Language: C90, C99
CERT C: DCL13-C

See Also

Topics

“Avoid Violations of MISRA C 2012 Rules 8.x”
“Check for Coding Rule 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 De nition

The restrict type quali er 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 Speci cation

The rule checker flags all uses of the `restrict` qualifier.

Message in Report

The restrict type qualifier shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C99
CERT C: EXP43-C
ISO/IEC TS 17961 ID: `restrict`

See Also

Topics

“Check for Coding Rule 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 Specification

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.

Message in Report

The value of an object with automatic storage duration shall not be read before it has been set.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Check Information

Group: Initialization
Category: Mandatory
AGC Category: Mandatory
Language: C90, C99
ISO/IEC TS 17961 ID: uninitref

See Also

MISRA C:2012 Rule 15.1 | MISRA C:2012 Rule 15.3

Topics

“Check for Coding Rule 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.

Message in Report

The initializer for an aggregate or union shall be enclosed in braces.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99

See Also

Topics

“Check for Coding Rule 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 De nition

Arrays shall not be partially initialized.

Rationale

Providing an explicit initialization for each array element makes it clear that every element has been considered.

Message in Report

Arrays shall not be partially initialized.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99

See Also

Topics

“Check for Coding Rule 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 De nition

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.

Message in Report

An element of an object shall not be initialized more than once.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C99

See Also

Topics

“Check for Coding Rule 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 De nition

Where designated initializers are used to initialize an array object the size of the array shall be speci ed 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.

Message in Report

Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C99
CERT C: ARR02-C

See Also

Topics

“Check for Coding Rule 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

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`) If you define a boolean type through a `typedef`, you must specify this type name before coding rules checking. For more information, see `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`

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	oating
[ ]	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					
== !=	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.

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 Rule 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
Language: C90, C99
CERT C: EXP46-C, INT02-C, INT07-C, INT13-C, INT16-C, INT31-C, STR04-C

See Also

MISRA C:2012 Rule 10.2

Topics

“Check for Coding Rule Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

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 character data arithmetically because the data does not represent numeric values.

For information on essential types, see `MISRA C:2012 Rule 10.1`.

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 Rule Violations Not Displayed”.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory
Language: C90, C99
CERT C: STR04-C

See Also

MISRA C:2012 Rule 10.1

Topics

“Check for Coding Rule Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

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.

For information on essential types, see `MISRA C:2012 Rule 10.1`.

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 Rule Violations Not Displayed”.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory

Language: C90, C99
CERT C: INT02-C, INT07-C, INT31-C, FLP06-C, STR04-C
ISO/IEC TS 17961 ID: intoflow

See Also

MISRA C:2012 Rule 10.4 | MISRA C:2012 Rule 10.5 | MISRA C:2012 Rule 10.6

Topics

“Check for Coding Rule Violations”
“Justify Coding Rule Violations Using Code Prover Checks” (Polyspace Code Prover)
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

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.

For information on essential types, see `MISRA C:2012 Rule 10.1`.

Polyspace Specification

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.

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 Rule Violations Not Displayed”.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory
Language: C90, C99
CERT C: INT02-C, INT07-C, INT18-C, INT31-C, STR04-C
ISO/IEC TS 17961 ID: `intoflow`

See Also

`MISRA C:2012 Rule 10.3` | `MISRA C:2012 Rule 10.7`

Topics

“Check for Coding Rule Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 10.5

The value of an expression should not be cast to an inappropriate essential type

Description

Rule De nition

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	oating
To	Boolean		Avoid	Avoid	Avoid	Avoid	Avoid
	character	Avoid					Avoid
	enum	Avoid	Avoid	Avoid	Avoid	Avoid	Avoid
	signed	Avoid					
	unsigned	Avoid					
	oating	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.

For more information on essential types, see `MISRA C:2012 Rule 10.1`.

Message in Report

The value of an expression should not be cast to an inappropriate essential type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Check Information

Group: The Essential Type Model
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 10.3` | `MISRA C:2012 Rule 10.8`

Topics

“Check for Coding Rule Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

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.

For information on essential types, see `MISRA C:2012 Rule 10.1`.

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 Rule Violations Not Displayed”.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory
Language: C90, C99
CERT C: INT02-C, INT18-C, INT31-C

See Also

MISRA C:2012 Rule 10.3 | MISRA C:2012 Rule 10.7

Topics

“Check for Coding Rule Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

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 De nition

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.

For information on essential types, see `MISRA C:2012 Rule 10.1`.

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 Rule Violations Not Displayed”.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory
Language: C90, C99
CERT C: INT02-C, INT18-C, INT31-C

See Also

Topics

“Check for Coding Rule Violations”
“Polyspace MISRA C:2012 Checkers”
“Software Quality Objective Subsets (C:2012)”

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 Specification

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 */
```

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 Rule 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
Language: C90, C99
CERT C: INT02-C
ISO/IEC TS 17961 ID: `ptrcomp`

See Also

`MISRA C:2012 Rule 10.5`

Topics

“Check for Coding Rule 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 Specification

Polyspace considers both explicit and implicit casts when checking this rule. However, casts from `NULL` or `(void*)0` do not violate this rule.

Message in Report

Conversions shall not be performed between a pointer to a function and any other type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99
CERT C: DCL07-C, EXP36-C, EXP37-C

See Also

Topics

“Check for Coding Rule 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.

Message in Report

Conversions shall not be performed between a pointer to an incomplete type and any other type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99
CERT C: EXP36-C
ISO/IEC TS 17961 ID: ptrcomp

See Also

MISRA C:2012 Rule 11.5

Topics

“Check for Coding Rule 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`

Message in Report

A cast shall not be performed between a pointer to object type and a pointer to a different object type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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 Quali er

```
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
Language: C90, C99
CERT C: EXP36-C, EXP39-C
ISO/IEC TS 17961 ID: `alignconv, ptrcomp`

See Also

`MISRA C:2012 Rule 11.4` | `MISRA C:2012 Rule 11.5` | `MISRA C:2012 Rule 11.8`

Topics

“Check for Coding Rule 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 Specification

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Message in Report

A conversion should not be performed between a pointer to object and an integer type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99
CERT C: INT36-C
ISO/IEC TS 17961 ID: intptrconv

See Also

MISRA C:2012 Rule 11.3 | MISRA C:2012 Rule 11.7 | MISRA C:2012 Rule 11.9

Topics

“Check for Coding Rule 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 Specification

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Message in Report

A conversion should not be performed from pointer to void into pointer to object.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99
CERT C: EXP36-C

See Also

`MISRA C:2012 Rule 11.2` | `MISRA C:2012 Rule 11.3`

Topics

“Check for Coding Rule 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 Specification

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Message in Report

A cast shall not be performed between pointer to void and an arithmetic type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99

See Also

Topics

“Check for Coding Rule 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 De nition

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.

Message in Report

A cast shall not be performed between pointer to object and a non-integer arithmetic type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99
CERT C: EXP36-C

See Also

MISRA C:2012 Rule 11.4

Topics

“Check for Coding Rule 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 Specification

Polyspace flags both implicit and explicit conversions that violate this rule.

Message in Report

A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99
CERT C: EXP05-C, EXP32-C

See Also

MISRA C:2012 Rule 11.3

Topics

“Check for Coding Rule 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 De nition

The macro NULL shall be the only permitted form of integer null pointer constant.

Rationale

The following expressions require 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.

Message in Report

The macro NULL shall be the only permitted form of integer null pointer constant.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99

See Also

MISRA C:2012 Rule 11.4

Topics

“Check for Coding Rule 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

Description	Operator and Operand	Precedence
Bitwise XOR	^	6
Bitwise OR	|	5
Logical AND	&&	4
Logical OR	||	3
Conditional	?:	2
Assignment	= *= /= += -= <<= >>= &= ^= |=	1
Comma	,	0

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 Rule 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.

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.

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
Language: C90, C99
CERT C: EXP00-C

See Also

MISRA C:2012 Rule 12.2 | MISRA C:2012 Rule 12.3 | MISRA C:2012 Rule 12.4

Topics

“Check for Coding Rule 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 Specification

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.

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 Rule Violations Not Displayed”.

Check Information

Group: Expressions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 12.1

Topics

“Check for Coding Rule 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 De nition

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.

Message in Report

The comma operator should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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.

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.
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.

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
Language: C90, C99

See Also

MISRA C:2012 Rule 12.1

Topics

“Check for Coding Rule 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 De nition

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.

Message in Report

Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule Violations Not Displayed”.

Check Information

Group: Expressions
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 12.1

Topics

“Check for Coding Rule 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.

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 Rule 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
Language: C90, C99
CERT C: EXP01-C, ARR01-C
ISO/IEC TS 17961 ID: `sizeofptr`

See Also

Topics

“Check for Coding Rule 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.

Message in Report

Initializer lists shall not contain persistent side effects.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C99

See Also

MISRA C:2012 Rule 13.2

Topics

“Check for Coding Rule 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

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.

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 Rule 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: Side Effects
Category: Required
AGC Category: Required
Language: C90, C99
CERT C: PRE31-C, EXP10-C, EXP30-C

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

Topics

“Check for Coding Rule 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.

Message in Report

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.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99

See Also

MISRA C:2012 Rule 13.2

Topics

“Check for Coding Rule 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 De nition

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`.

Message in Report

The result of an assignment operator should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99
ISO/IEC TS 17961 ID: boolasgn

See Also

MISRA C:2012 Rule 13.2

Topics

"Check for Coding Rule 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 Specification

- For this rule, Polyspace considers that all function calls have a persistent side effect.
- If the right operand is a volatile variable, Polyspace does not flag this as a rule violation.

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 Rule Violations Not Displayed”.

Examples

Right Operand of Logical Operator with Persistent Side E ects

```
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
Language: C90, C99

See Also

Topics

"Check for Coding Rule 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 De nition

The operand of the sizeof operator shall not contain any expression which has potential side e ects.

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 Speci cation

The rule is not violated if the argument is a `volatile` variable.

Message in Report

The operand of the sizeof operator shall not contain any expression which has potential side effects.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99
CERT C: EXP44-C

See Also

MISRA C:2012 Rule 18.8

Topics

“Check for Coding Rule 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 Specification

If the `for` index is a variable symbol, Polyspace checks that it is not a float.

Message in Report

A loop counter shall not have essentially floating type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99
CERT C: FLP30-C

See Also

MISRA C:2012 Rule 14.2

Topics

“Check for Coding Rule 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 Specification

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.

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 Rule 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.

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
Language: C90, C99

See Also

MISRA C:2012 Rule 14.1 | MISRA C:2012 Rule 14.3 | MISRA C:2012 Rule 14.4

Topics

“Check for Coding Rule 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 Specification

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.

Message in Report

- Boolean operations whose results are invariant shall not be permitted.
- Expression is always true.
- Boolean operations whose results are invariant shall not be permitted.
- 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 Rule Violations Not Displayed”.

Check Information

Group: Control Statement Expressions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 2.1 | MISRA C:2012 Rule 14.2

Topics

“Check for Coding Rule 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 Specification

Polyspace does not flag integer constants, for example `if(2)`.

If your configuration includes the option `-boolean-types`, the number of warnings can increase or decrease.

Message in Report

The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99

See Also

MISRA C:2012 Rule 14.2 | MISRA C:2012 Rule 20.8

Topics

“Check for Coding Rule 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.

Message in Report

The `goto` statement should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99

See Also

MISRA C:2012 Rule 15.2 | MISRA C:2012 Rule 15.3 | MISRA C:2012 Rule 15.4

Topics

“Check for Coding Rule 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.

Message in Report

The goto statement shall jump to a label declared later in the same function.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99

See Also

`MISRA C:2012 Rule 15.1` | `MISRA C:2012 Rule 15.3` | `MISRA C:2012 Rule 15.4`

Topics

“Check for Coding Rule 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.

Message in Report

Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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
Language: C90, C99

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`

Topics

“Check for Coding Rule 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 De nition

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.

Message in Report

There should be no more than one break or goto statement used to terminate any iteration statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to “Coding Rule 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.