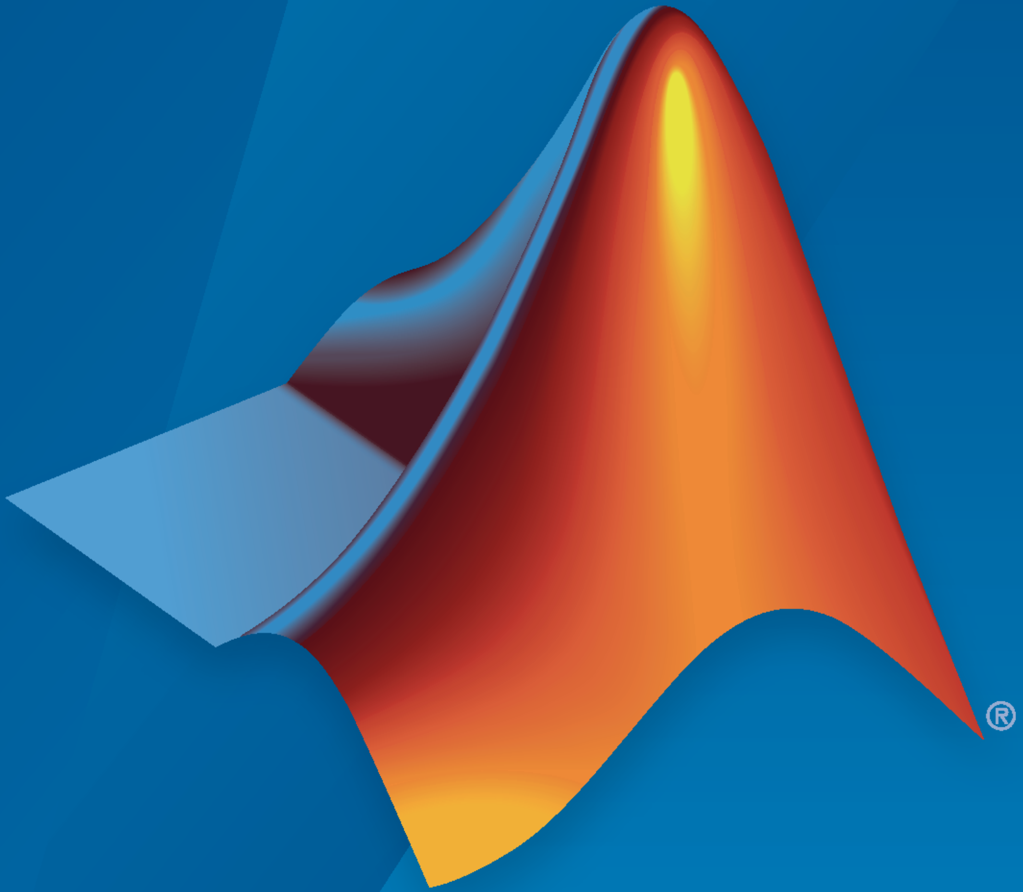
Polyspace® Bug Finder™ Server™ Release Notes



MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

Polyspace® Bug Finder™ Server™ Release Notes

Contents

R2021a

R2021a

Review Information Between Polyspace Access Projects: Download results from Polyspace Access and import review information to a different project

Summary: In R2021a, you can merge review information between Polyspace® Access projects.

To merge the review information between projects:

1. Use the `polyspace-access -download` command to download the results from the project where you already reviewed the findings.
2. Use the `-import-comments` option to import the review information as you run an analysis on the other project.
3. Upload the analysis results containing the imported review information to Polyspace Access.

See `polyspace-access`.

Benefits: If you review findings in one project and you reuse the source code that contains those findings in another project, you do not need to review those findings again.

Configuration from Build System: Specify options delimiter and suppress console output

Summary: In R2021a, `polyspace-configure` has new options to simplify the creation of a Polyspace project or options file:

- `-options-for-sources-delimiter` — Use this option to specify an ASCII character that Polyspace uses as a delimiter between a group of analysis options. You typically use this option in combination with `-options-for-sources`, which associates a group of analysis options with specific source files. You might want to specify a delimiter if, for instance, the default delimiter (`;`) is already used inside a macro.
- `-no-console-output` — Use this option to completely suppress the console output of `polyspace-configure`, including error and warning messages. By default, `polyspace-configure` emits errors and warnings only.

See also `polyspace-configure`.

Benefits: The new options allow you to customize the `polyspace-configure` runs without extensive additional scripting.

Configuration from Build System: Improved detection of incompatible software

Summary: In R2021a, if you use software that is not compatible with `polyspace-configure` when you trace your build process, `polyspace-configure` emits a message that identifies the software and that provides contextual help if applicable. Software that is not compatible with `polyspace-configure` includes some antivirus software and certain build systems such as Bazel.

For more information, see `polyspace-configure`.

Benefits: Previously, when `polyspace-configure` could not trace your build process because of incompatible software, the command output did not identify the software. Now, you can easily check if your build system and environment is compatible with `polyspace-configure`.

Updated GCC Compiler Support: Set up Polyspace analysis for code compiled with GCC version 8.x

Summary: In R2021a, Polyspace supports the GCC compiler version 8.x natively. If you build your source code by using GCC version 8.x, you can specify the compiler name for your Polyspace analysis.

You specify a compiler in the command line by using the option `Compiler (-compiler)`.

```
polyspace-bug-finder-server -compiler gnu8.x -sources file.c ...
```

Benefits: Because of the native support, you can now set up a Polyspace project without knowing the internal workings of this compiler. The analysis can interpret macros that are implicitly defined by the compiler and compiler-specific language extensions such as keywords and pragmas.

Updated Microsoft Visual C++ Support: Set up a Polyspace analysis for code compiled with Visual Studio 2019

Summary: In R2021a, Polyspace supports the compiler Visual Studio® 2019 natively. If you build your source code by using Visual Studio 2019 (versions 16.x), you can specify the compiler name for your Polyspace analysis.

You specify a compiler in the command line by using the option `Compiler (-compiler)`.

```
polyspace-bug-finder-server -compiler visual16.x -sources file.c ...
```

Benefits: Because of the native support, you can now set up a Polyspace project without knowing the internal workings of this compiler. The analysis can interpret macros that are implicitly defined by the compiler and compiler-specific language extensions such as keywords and pragmas.

AUTOSAR C++14 Support: Check for 327 AUTOSAR C++14 rules including 19 new rules in R2021a

Summary: In R2021a, you can look for violations of these AUTOSAR C++14 rules in addition to previously supported rules.

AUTOSAR C++14 Rule	Description	Polyspace Checker
A2-7-3	All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation.	`AUTOSAR C++14 Rule A2-7-3`
A2-8-1	A header file name should reflect the logical entity for which it provides declarations.	`AUTOSAR C++14 Rule A2-8-1`

AUTOSAR C++14 Rule	Description	Polyspace Checker
A2-8-2	An implementation file name should reflect the logical entity for which it provides definitions.	`AUTOSAR C++14 Rule A2-8-2`
A8-4-5	"consume" parameters declared as X && shall always be moved from.	`AUTOSAR C++14 Rule A8-4-5`
A8-4-6	"forward" parameters declared as T && shall always be forwarded.	`AUTOSAR C++14 Rule A8-4-6`
A8-4-8	Output parameters shall not be used.	`AUTOSAR C++14 Rule A8-4-8`
A8-4-9	"in-out" parameters declared as T & shall be modified.	`AUTOSAR C++14 Rule A8-4-9`
A8-4-10	A parameter shall be passed by reference if it can't be NULL.	`AUTOSAR C++14 Rule A8-4-10`
A8-5-4	If a class has a user-declared constructor that takes a parameter of type std::initializer_list, then it shall be the only constructor apart from special member function constructors.	`AUTOSAR C++14 Rule A8-5-4`
A12-8-1	Move and copy constructors shall move and respectively copy base classes and data members of a class, without any side effects.	`AUTOSAR C++14 Rule A12-8-1`
A12-8-2	User-defined copy and move assignment operators should use user-defined no-throw swap function.	`AUTOSAR C++14 Rule A12-8-2`
A12-8-3	Moved-from object shall not be read-accessed.	`AUTOSAR C++14 Rule A12-8-3`
A13-5-3	User-defined conversion operators should not be used.	`AUTOSAR C++14 Rule A13-5-3`
A13-6-1	Digit sequences separators ' shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4 digits.	`AUTOSAR C++14 Rule A13-6-1`
A15-4-1	Dynamic exception-specification shall not be used.	`AUTOSAR C++14 Rule A15-4-1`
A15-4-4	A declaration of non-throwing function shall contain noexcept specification.	`AUTOSAR C++14 Rule A15-4-4`

AUTOSAR C++14 Rule	Description	Polyspace Checker
A20-8-1	An already-owned pointer value shall not be stored in an unrelated smart pointer.	`AUTOSAR C++14 Rule A20-8-1`
A27-0-4	C-style strings shall not be used.	`AUTOSAR C++14 Rule A27-0-4`
M5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	`AUTOSAR C++14 Rule M5-0-16`

See also “AUTOSAR C++14 Rules” (Polyspace Bug Finder Access).

CERT C++ Support: Check for memory management and programming rule violations.

Summary: In R2021a, you can look for violations of these CERT C++ rules in addition to previously supported rules.

CERT C++ Rule	Description	Polyspace Checker
OOP50-CPP	Do not invoke virtual functions from constructors or destructors	`CERT C++: OOP50-CPP`
EXP63-CPP	Do not rely on the value of a moved-from object	`CERT C++: EXP63-CPP`
MEM56-CPP	Do not store an already-owned pointer value in an unrelated smart pointer	`CERT C++: MEM56-CPP`

See also “CERT C++ Rules” (Polyspace Bug Finder Access).

MISRA C++:2008 Support: Check for disallowed pointer arithmetic

Summary: In R2021a, you can look for violation of this MISRA C++:2008 rule in addition to previously supported rules.

Rule	Description	Polyspace Checker
MISRA C++:2008 Rule 5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	`MISRA C++:2008 Rule 5-0-16`

See also “MISRA C++:2008 Rules” (Polyspace Bug Finder Access).

MISRA C:2012 Support: Checkers updated to account for MISRA C:2012 Technical Corrigendum 1 and Amendment 2

Summary: In R2021a, Polyspace supports amendments to MISRA C®:2012 rules in Technical Corrigendum 1 and Amendment 2.

MISRA C:2012 Technical Corrigendum 1

MISRA C:2012 Technical Corrigendum 1 adds clarifications to existing rules. The clarifications have led to changes in these checkers:

Rule	Description	Update in Technical Corrigendum 1
`MISRA C:2012 Rule 10.1`	Operands shall not be of an inappropriate essential type.	The rule now explicitly forbids use of pointer types with logical operands such as &&, `||` and `!`.
`MISRA C:2012 Rule 10.5`	The value of an expression should not be cast to an inappropriate essential type.	The rule now forbids casts of integer constants with value 0 or 1 to essentially enum types.
`MISRA C:2012 Rule 11.2`	Conversions shall not be performed between a pointer to an incomplete type and any other type.	The rule now takes into account only the unqualified types that the pointers point to. For instance, if a pointer is assigned to another and the only difference between the pointed types is a `const` qualifier, the rule does not consider this assignment as a conversion.
`MISRA C:2012 Rule 11.4`	A conversion should not be performed between a pointer to object and an integer type.	The rule now applies explicitly to pointers to objects only. Conversions between an integer type and other pointer types such as `void*` or pointers to functions are flagged by other rules.
`MISRA C:2012 Rule 11.9`	The macro NULL shall be the only permitted form of integer null pointer constant.	The rule allows the use of `{0}` to initialize aggregates or unions containing pointers.
`MISRA C:2012 Rule 14.2`	A for loop shall be well-formed.	The rule allows any form of initialization of the loop counter as long as the initialization does not have other side effects.

MISRA C:2012 Amendment 2

MISRA C:2012 Amendment 2 addresses the new language features in the C11 standard. All updates in Amendment 2 have been incorporated in the checkers.

Rule	Description	Update in Amendment 2
`MISRA C:2012 Rule 1.4`	Emergent language features shall not be used.	This rule is new in Amendment 2.
`MISRA C:2012 Rule 12.1`	The precedence of operators within expressions should be made explicit.	The rule now mandates a violation if the operand of the `_Alignof` operator is not enclosed in parenthesis.
`MISRA C:2012 Rule 21.3`	The memory allocation and deallocation functions of `<stdlib.h>` shall not be used.	The rule now flags uses of the `aligned_alloc` function.
`MISRA C:2012 Rule 21.8`	The Standard Library termination functions of <stdlib.h> shall not be used.	The rule no longer flags `system`. In addition to `exit` and `abort`, the rule now flags `_Exit` and `quick_exit`.
`MISRA C:2012 Rule 21.21`	The Standard Library function `system` of `<stdlib.h>` shall not be used.	This rule is new in Amendment 2.
`MISRA C:2012 Rule 22.1`	All resources obtained dynamically by means of Standard Library functions shall be explicitly released.	The rule now flags memory allocation using the `aligned_alloc` function if the memory is not released.

Modifying Checker Behavior: Modify parameters for MISRA C:2012 rules 1.1 and 5.1 to 5.5

Summary: In R2021a, you can modify the thresholds used in the checkers for MISRA C: 2012 Rules 1.1 and 5.1 to 5.5.

Rule	Description	Supported Modification
`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.	You can increase or decrease these parameters of the rule checker: • Maximum depth of nesting allowed in control flow statements • Maximum levels of inclusion allowed using include files • Maximum number of constants allowed in an enumeration • Maximum number of macros allowed in a translation unit • Maximum number of members allowed in a structure • Maximum levels of nesting allowed in a structure

`MISRA C:2012 Rule 5.1`
`MISRA C:2012 Rule 5.2`
`MISRA C:2012 Rule 5.3`
`MISRA C:2012 Rule 5.4`
`MISRA C:2012 Rule 5.5`
	These rules require uniqueness of certain types of identifiers. For instance, rule 5.1 requires that external identifiers be distinct.	If the difference between two identifiers occurs beyond the first *`num`* characters, the rule checker considers the identifiers as identical. You can modify the parameter *`num`* separately for external and internal identifiers.

For more information, see:

- "Modify Default Behavior of Bug Finder Checkers"
- `-code-behavior-specifications`

Benefits: You can adapt the checkers for MISRA C: 2012 Rules 1.1 and 5.1 to 5.5 to follow your compiler specifications.

Guidelines: New checkers for software complexity defects

Summary: In R2021a, Polyspace has a new category of checkers called **Guidelines**. This category contains the **Software Complexity** checkers. Reduce the software complexity metrics of your code by activating these new checkers. See "Reduce Software Complexity by Using Polyspace Checkers". The **Software Complexity** checkers include:

Defect	Description
`Number of Calling Functions Exceeds Threshold`	The number of distinct callers of a function is greater than the defined threshold.

Defect	Description
`Number of Called Functions Exceeds Threshold`	The number of distinct function calls within the body of a function is greater than the defined threshold.
`Comment density below threshold`	The comment density of the module falls below the specified threshold.
`Call Tree Complexity Exceeds Threshold`	The call tree complexity of a file is greater than the defined threshold.
`Number of Lines Within body Exceeds Threshold`	The number of lines in the body of a function is greater than the defined threshold.
`Number of Executable Lines Exceeds Threshold`	The number of executable lines in the body of a function is greater than the defined threshold.
`Number of GOTO Statements Exceeds Threshold`	The nesting depth of control structures in a function is greater than the defined nesting depth threshold of a function.
`Number of Call Levels Exceeds Threshold`	The number of `goto` statements in a function is greater than the defined threshold.
`Number of Local Static variables Exceeds Threshold`	The number of local static variables in a function is greater than the defined threshold.
`Number of Local Nonstatic Variables Exceeds Threshold`	The number of function calls in a function is greater than the defined call occurrence threshold of a function.
`Number of Call Occurrences Exceeds Threshold`	The number of function calls in a function is greater than the defined call occurrence threshold of a function.
`Number of Function Parameters Exceeds Threshold`	The number of arguments of a function is greater than the defined threshold.
`Number of Paths Exceeds Threshold`	The number of static paths in a function is greater than the defined threshold.
`Number of Return Statements Exceeds Threshold`	The number of `return` statements in a function is greater than the defined threshold.
`Number of Instructions Exceeds Threshold`	The number of instructions in a function is greater than the defined threshold.
`Number of Lines Exceeds Threshold`	The number of total lines in a file is greater than the defined threshold.
`Cyclomatic Complexity Exceeds Threshold`	The cyclomatic complexity of a function is greater than the defined cyclomatic complexity threshold of a function.
`Language Scope Exceeds Threshold`	The language scope of a function is greater than the defined threshold.

In the Polyspace user interface, activate these checkers in the **Coding Standard & Code Metric** node of the **Configuration** pane. Alternatively, in the Checkers selection window, select the **Guidelines > Software Complexity** checkers.

To activate these checkers in the command-line, use the analysis option `Check Guidelines (-guidelines)`. To specify a subset of these checkers with modified thresholds by using a checkers selection file, use `Set checkers by file (-checkers-selection-file)`.

Compatibility Considerations

Each of these software complexity checkers corresponds to a code metric. When you import comments from a previous run by using the command `polyspace-comments-import`, Polyspace copies any review information on a code metric in the previous result to the corresponding software complexity checker in the current result. If the current result contains the same code metric, the review information is also copied to the code metric.

JSF AV C++ Support: Check for cases where pass-by-reference is preferred to pass-by-pointer

Summary: In R2021a, you can check for this JSF® AV C++ rule in addition to previously supported rules.

Rule	Description
AV Rule 117	Arguments **should** be passed by reference if NULL values are not possible.

See also “JSF AV C++ Coding Rules” (Polyspace Bug Finder Access).

New Bug Finder Checkers: Check for inefficient string operations, noncompliance with AUTOSAR Standard specifications, and other issues

Summary: In R2021a, you can check for these new Bug Finder defects in your code.

Defect	Description
`Const rvalue reference parameter may cause unnecessary data copies`	The `const`-ness of an rvalue reference prevents move operation and causes a more expensive copy operation instead.
`Expensive use of std::string methods instead of more efficient overload`	An `std::string` method uses a single character string literal, that is, a `const char*` object of length one, instead of using a single quoted character.
`Expensive use of std::string with empty string literal`	Use of `std::string` with empty string literal can be replaced by less expensive calls to `std::basic_string` member functions.
`Expensive use of non-member std::string operator+() instead of a simple append`	The non-member `std::string operator+()` function is called when the append (or `+=`) method would have been more efficient.
`Expensive local variable copy`	A local variable is created by copy from a `const` reference and not modified later.

Defect	Description
`Expensive logical operation`	A logical operation requires the evaluation of both operands because of their order, resulting in inefficient code.
`File does not compile`	A file has a compilation error.
`Noncompliance with AUTOSAR library`	A call to an AUTOSAR RTE API function violates AUTOSAR Standard specifications.
`Use of new or make_unique instead of more efficient make_shared`	Creating a `shared_ptr` pointer with `new` or `make_unique` causes an unnecessary additional memory allocation.

For all defect checkers, see “Defects” (Polyspace Bug Finder Access).

Changes in analysis options and binaries

-code-behavior-specifications takes only one file as argument
Behavior change

Starting in R2021a, this option only takes one XML file as argument. If you were specifying code behaviors in multiple XML files, combine their content into one file and provide this file as argument to the option.

See also `-code-behavior-specifications`.

-sources-encoding with value other than auto disables automatic detection of encoding
Behavior change

Starting in R2021a, if you explicitly specify a value with the option `-sources-encoding` (or use the default value `system` which uses the default encoding of your OS), the analysis does not perform any automatic detection of source file encoding. For instance, if you use `-sources-encoding shift-jis`, the analysis internally converts your source files from Shift JIS (Shift Japanese Industrial Standards) to UTF-8 encoding before processing them. If you see regressions from previous releases, consider using `-sources-encoding auto` to reenable the automatic detection of source encoding. Automatic detection is useful when your project contains, for instance, a mix of different encodings.

See also `Source code encoding (-sources-encoding)`.

Changes to coding rules checking

In R2021a, coding rules checking has improved across various standards. For instance, you can check for both MISRA C:2004 and MISRA C:2012 rules in the same run.

These changes have been made in the checking of previously supported rules.

Rule	Description	Change
`MISRA C:2012 Rule 1.1`	An implementation-defined behavior on which the output of the program depends shall be documented and understood.	You can now change the thresholds used in the rule checking using the option `-code-behavior-specifications`.

Rule	Description	Change
`MISRA C:2012 Rule 1.3`	There shall be no occurrence of undefined or critical unspecified behaviour.	The checker no longer flags all instances of the `offsetof` macro, but only the instances that cause undefined behavior. For instance, if the second argument of `offsetof` is not a field of the first argument or is a bitfield, the checker raises a violation.
MISRA C:2012 Rule 5.x (Polyspace Bug Finder Access)	Rules that ensure uniqueness of identifiers.	You can now change the thresholds used in the rule checking using the option `-code-behavior-specifications`.
`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.	The checker now detects implicit conversions when a structure is initialized using aggregate initialization. For instance, in this code snippet, the initialization of structure `a` results in an implicit conversion from the essentially signed type of `x` to the essentially unsigned type of the field `a_` `typedef struct tag_A` `{` `unsigned char a_;` `unsigned char b_;` `}tag_A;` `int x = 1;` `tag_A a = {x,0}; //Noncompliant`
`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.	The checker treats macros such as TRUE or FALSE that resolve to 0 or 1 as essentially boolean.
`AUTOSAR C++14 Rule A5-0-3`	The declaration of objects shall contain no more than two levels of pointer indirection	The checker no longer flags the use of objects with more than two levels of pointer indirection.

Rule	Description	Change
`AUTOSAR C++14 Rule A8-5-2`	Braced-initialization {}, without equals sign, shall be used for variable initialization.	The checker now adheres more strictly to the AUTOSAR C++14 specifications. The checker flags non-uniform initializations such as: • `*Type* obj1 = obj2;` • `*Type* obj1(obj2);` Even if `obj1` and `obj2` have the same types. Previously, the checker raised a flag only if the types were different. The checker allows an exception for these cases: • Initialization of variables with type `auto` using a simple assignment • Initialization of reference types • Declarations with global scope using the format *`Type`* `a()` where *`Type`* is a class type with default constructor. The analysis interprets `a` as a function returning the type *`Type`*. • Loop variable initialization in OpenMP parallel `for` loops, that is, in `for` loop statements that immediately follow `#pragma omp parallel for`
`AUTOSAR C++14 Rule A10-1-1`	Classes shall not be derived from more than one base class which is not an interface class.	The checker now expands interface classes to include constructors and destructors set to `=default` or `=delete`.
`AUTOSAR C++14 Rule A12-6-1`	All class data members that are initialized by the constructor shall be initialized using member initializers.	The checker no longer flags constructors that use default member initialization.
`AUTOSAR C++14 Rule A12-8-7`	Assignment operators should be declared with the ref-qualifier &.	The checker no longer flags deleted assignment operators without the ref-qualifier &.

Rule	Description	Change
`AUTOSAR C++14 Rule A20-8-5` and `AUTOSAR C++14 Rule A20-8-6`	`std::make_unique` (`std::make_shared`) shall be used to construct objects owned by `std::unique_ptr` (`std::shared_ptr`).	The checkers now also apply to `boost::unique_ptr` and `boost::shared_ptr`.
`AUTOSAR C++14 Rule M5-0-15`	Array indexing shall be the only form of pointer arithmetic.	The checker no longer flags arithmetic operations such as increment and decrement on iterators that point to elements in containers.
`CERT C: Rule INT31-C`	Ensure that integer conversions do not result in lost or misinterpreted data	The checker now detects comparisons of `time_t` variables with variables of other types. `time_t` is an implementation-defined type, therefore, these comparisons can lead to unexpected results.
`CERT C: Rec. MEM04-C`	Beware of zero-length allocations	The checker now performs more direct checks for possibilities of zero-length memory allocations and adheres more strictly to the CERT C standard.
`CERT C++: DCL53-CPP`	Do not write syntactically ambiguous declarations.	The checker no longer flags ambiguous declarations with global scope. For instance, the analysis does not flag declarations with global scope using the format *`Type`* `a()` where *`Type`* is a class type with a default constructor. The analysis interprets `a` as a function returning the type *`Type`*.
`CERT C: Rule EXP37-C`	Call functions with the correct number and type of arguments	This checker now flags: • Calls with complex arguments to math functions that do not take a complex input • Calls to functions whose provided or deduced prototypes do not match their definitions.
`CERT C++: EXP37-C`	Call functions with the correct number and type of arguments	This checker now flags the calls to an `extern "C"` function if its prototypes does not match the definition.

Rule	Description	Change
Checkers from different C++ standards: • `MISRA C++:2008 Rule 3-2-2` • `AUTOSAR C++14 Rule M3-2-2` • `CERT C++: DCL60-CPP`	Checkers for the one definitions rule in C++.	Starting in R2021a, in the declarations that violate these rules, the violations are flagged on the keywords instead of the variable names. Starting in R2021a, these checkers are no longer raised on unused code such as: • Noninstantiated templates • Uncalled `static` or `extern` functions • Uncalled and undefined local functions • Unused types and variables
Checkers from different C++ standards: • `MISRA C++:2008 Rule 3-2-1` • `AUTOSAR C++14 Rule M3-2-1`	Checkers that flag declaration of an object with incompatible types across modules.	Starting in R2021a, Polyspace considers two types to be compatible if they have the same size and signedness in the environment that you use. For instance, if you specify `-target` as i386, Polyspace considers `long` and `int` to be compatible types. Starting in R2021a, these checkers are no longer raised on unused code such as: • Noninstantiated templates • Uncalled `static` or `extern` functions • Uncalled and undefined local functions • Unused types and variables

Rule	Description	Change
Checkers from different C++ standards: • MISRA C++2008: • `MISRA C++:2008 Rule 2-10-5` • `MISRA C++:2008 Rule 3-2-4` • AUTOSAR C++14: • `AUTOSAR C++14 Rule A2-10-4` • `AUTOSAR C++14 Rule A2-10-5` • `AUTOSAR C++14 Rule M3-2-4`	Checkers that check for inconsistent declaration and definitions, and name re-use across different modules.	Starting in R2021a, these checkers are no longer raised on unused code such as: • Noninstantiated templates • Uncalled `static` or `extern` functions • Uncalled and undefined local functions • Unused types and variables
`JSF AV C++ Rule 137`		Starting in R2021a, this checker is raised on declarations of nonstatic objects that you use in only one file. The checker is raised even if you analyze a singe file. The checker is not raised on the declarations of objects that remain unused, such as: • Noninstantiated templates • Uncalled `static` or `extern` functions • Uncalled and undefined local functions • Unused types and variables

Compatibility Considerations

If you checked your code for the preceding rules, you might see a change in the number of violations.

Updated Bug Finder defect checkers

In R2021a, these defect checkers have been updated.

Defect	Description	Update
`Ambiguous declaration syntax`	Declaration syntax can be interpreted as object declaration or part of function declaration	The checker no longer flags ambiguous declarations with global scope. For instance, the analysis does not flag declarations with global scope using the format *`Type`* `a()` where *`Type`* is a class type with a default constructor. The analysis interprets `a` as a function returning the type *`Type`*.
`Format string specifiers and arguments mismatch`	Format specifiers in `printf`-like functions do not match corresponding arguments	In cases where integer promotion modifies the perceived data type of an argument, the analysis result shows both the original type and the type after promotion. The format specifier has to match the type after integer promotion.

R2020b

Compiler Support: Set up Polyspace analysis for code compiled with Renesas SH C compilers

Summary: If you build your source code by using Renesas® SH C compilers, in R2020b, you can specify the target name `sh`, which corresponds to SuperH targets, for your Polyspace analysis.

You specify a compiler and target by using the options `Compiler (-compiler)` and `Target processor type (-target)`.

```
polyspace-bug-finder-server -compiler renesas -target sh -sources file.c....
```

See also `Renesas Compiler (-compiler renesas)`.

Benefits: You can now set up a Polyspace project without knowing the internal workings of Renesas SH C compilers. If your code compiles with your compiler, it will compile with Polyspace in most cases without requiring additional setup. Previously, you had to explicitly define macros that were implicitly defined by the compiler and remove unknown language extensions from your preprocessed code.

Cygwin Support: Create Polyspace projects automatically by using Cygwin 3.x build commands

Summary: In R2020b, the `polyspace-configure` command supports version 3.x of Cygwin™ (versions 3.0, 3.1, and so on).

See also Check if Polyspace Supports Build Scripts.

Benefits: Using the `polyspace-configure` command, you can trace build scripts that are executed at a Cygwin 3.x command line and create a Polyspace project with the source files and compilation options automatically specified.

C++17 Support: Run Polyspace analysis on code with C++17 features

Summary: In R2020b, Polyspace can interpret the majority of C++17-specific features.

See also:

- `C++ standard version (-cpp-version)`
- C/C++ Language Standard Used in Polyspace Analysis
- C++17 Language Elements Supported in Polyspace

Benefits: You can now set up a Polyspace analysis for code containing C++17-specific language elements. Previously, some C++17 specific language elements were not recognized and caused compilation errors.

Configuration from Build System: Generate a project file or analysis options file by using a JSON compilation database

Summary: In R2020b, if your build system supports the generation of a JSON compilation database, you can create a Polyspace project file or an analysis options file from your build system without tracing your build process. After you generate the JSON compilation database file, pass this file to `polyspace-configure` by using the option `-compilation-database` to extract your build information.

For more information on compilation databases, see JSON Compilation Database.

Benefits: Previously, you had to invoke your build command and trace your build process to extract the build information. For some build systems such as Bazel, `polyspace-configure` could not always trace the build process, resulting in errors when running an analysis by using the generated options file.

Configuration from Build System: Specify how Polyspace imports compiler macro definitions

Summary: In R2020b, when you use `polyspace-configure` to create a Polyspace project file or to generate an analysis options file from your build system, you can specify how Polyspace imports the compiler macro definitions.

Use option `-import-macro-definitions` and specify:

- `none` — Skip the import of macro definition. You can provide macro definitions manually instead.
- `from-whitelist` — Use a Polyspace white list to query your compiler for macro definitions.
- `from-source-token` — Use all non-keyword tokens in your source files to query your compiler for macro definitions.

See also `polyspace-configure`.

Benefits: Previously, Polyspace used all non-keyword tokens in your source files to query your compiler for macro definitions each time that you traced your build command. You now have greater control on the import of macro definitions.

Configuration from Build System: Compiler configuration cached from prior runs for improved performance

Summary: In R2020b, when you use `polyspace-configure` to create a Polyspace project file or to generate an analysis options file from your build system, Polyspace caches your compiler configuration. If your compiler configuration does not change, Polyspace reuses the cached configuration during subsequent runs of `polyspace-configure`.

See also `polyspace-configure`.

Benefits: Previously, Polyspace did not cache your compiler configuration. Instead, during every run of `polyspace-configure`, Polyspace queried your compiler for the size of fundamental types, compiler macro definitions, and other compiler configuration information. Starting R2020b, the caching improves the later `polyspace-configure` runs.

Offloading Analysis: Submit Polyspace analysis jobs from CI server to a dedicated analysis cluster

Summary: In R2020b, you can set up a continuous integration (CI) system to offload a Polyspace analysis to a dedicated cluster and download the results after analysis. The cluster performing the analysis can be one server or several servers where a head node distributes the jobs to several worker nodes which perform the analysis. MATLAB® Parallel Server™ is required on all servers involved in distributing jobs or running the analysis.

See Offload Polyspace Analysis from Continuous Integration Server to Another Server.

Benefits: When running static code analysis with Polyspace as part of continuous integration, you might want the analysis to run on a server that is different from the server running your continuous integration (CI) scripts. For instance, you might want to perform the analysis on a server that has more processing power. You can then offload the analysis from your CI server to the other server.

Offloading Analysis: Server-side errors reported back to client side

Summary: If you run a Polyspace analysis on a MATLAB Parallel Server cluster, in R2020b, server-side errors are reported back in the client-side log.

The log contains this additional information reported back from the server side:

- Errors that occurred during the server-side analysis.

  For instance, if a Polyspace Server license has not been activated, you see a license checkout failure reported from the server side.
- Path to the Polyspace Server instance that runs the analysis.

Information reported from the server side appears in the log between the `Start Diary` and `End Diary` lines.

Benefits: Starting R2020b, you can troubleshoot server-side errors more easily by using the log reported on the client side.

Results Export: Export Polyspace results to external formats such as SARIF JSON

Summary: In R2020b, you can use the new `polyspace-results-export` command to export Polyspace results to formats such as JSON and CSV.

- The JSON object follows the Static Analysis Results Interchange Format or SARIF notation.
- The CSV file has the same fields as produced by using the earlier `polyspace-report-generator` command with the `-generate-results-list-file` option.

  Use the `polyspace-report-generator` command to generate PDF or Word reports in a predefined format. To package results using your own format, export them using the `polyspace-results-export` command and read the resulting JSON object or CSV file.

You can use this command with results generated locally or with results uploaded to Polyspace Access.

See also `polyspace-results-export`.

Benefits: Using the JSON object or CSV file, you can display results in a convenient format. For instance, you can group defects found by Bug Finder based on their impact. Because the JSON object follows a standard notation, you can also use this format to display Polyspace results with results from other tools.

User Authentication: Use a credentials file to pass your Polyspace Access credentials at the command line

Summary: In R2020b, if you use a command that requires your Polyspace Access credentials, you can save these credentials in a file that you pass to the command. If you use that command inside a script, you no longer need to store your credentials in the script.

To create a credentials file, enter a set of credentials, either as `-login` and `-encrypted-password` entries on separate lines, for example:

```
-login jsmith
-encrypted-password LAMMMEACDMKEFELKMNDCONEAPECEEKPL
```

Or as a `-api-key` entry:

```
-api-key keyValue123
```

For more information on generating API keys, see Configure **User Manager** (Polyspace Bug Finder Access) (Polyspace Code Prover Access).

Save the file and pass it to the command by using the `-credentials-file` flag. You can use the credentials file with these Polyspace commands:

- `polyspace-access`
- `polyspace-results-export`
- `polyspace-report-generator`

For increased security, restrict the read/write permissions for the credentials file.

Benefits: Previously, you could provide your Polyspace Access credentials in a script only by passing them directly to the command. Starting R2020b, when the command that requires the credentials runs, someone who is inspecting currently running processes, for instance, by using the command `ps aux` on Linux, can no longer see your credentials.

Importing Review Information: Accept information in source or destination results folder in case of merge conflicts

Summary: In R2020b, when importing review information such as severity, status, and comments at the command line, if the same result has different review information in the source and destination folder, you can choose one of the following:

- That the review information in the destination folder is retained.

  This behavior is the default behavior of the `polyspace-comments-import` command.
- That the review information in the source folder overwrites the information in the destination folder.

  You can switch to this behavior using the new option `-overwrite-destination-comments`.

See also `polyspace-comments-import`.

Benefits: Previously, newer review information in the destination folder was retained and could not be overwritten. Now, when merging review information, you can choose whether the source or destination folder takes precedence in case of merge conflicts.

polyspacePackNGo Function: Generate and package Polyspace option files from a Simulink model

Summary: In R2020b, you can package Polyspace option files along with code generated from a Simulink® model, and then analyze the code on a different machine in a distributed workflow. After packaging the generated code, create and archive options files required for a Polyspace analysis by using the `polyspacePackNGo` function.

See also:

- `polyspacePackNGo`
- (Polyspace Code Prover) (Simulink) (Polyspace Code Prover Server)Run Polyspace Analysis on Generated Code by Using Packaged Options Files (Polyspace Bug Finder Server)

Benefits: In a distributed workflow, a Simulink user generates code from a model and sends the code to another development environment. In this environment, a Polyspace user analyzes the generated code by using design ranges and other model-specific information. Previously, in this distributed workflow, you configured the Polyspace analysis options manually. Starting in R2020b, you do not have to manually create the option files when analyzing generated code by using Polyspace in a distributed workflow.

AUTOSAR C++14 Support: Check for 308 AUTOSAR C++14 rules including 61 new rules in R2020b

Summary: In R2020b, you can look for violations of these AUTOSAR C++14 rules in addition to previously supported rules.

AUTOSAR C++14 Rule	Description	Polyspace Checker
A0-1-1	A project shall not contain instances of non-volatile variables being given values that are not subsequently used.	`AUTOSAR C++14 Rule A0-1-1`
A0-1-3	Every function defined in an anonymous namespace, or static function with internal linkage, or private member function shall be used.	`AUTOSAR C++14 Rule A0-1-3`
A2-7-2	Sections of code shall not be "commented out".	`AUTOSAR C++14 Rule A2-7-2`
A2-10-4	The identifier name of a non-member object with static storage duration or static function shall not be reused within a namespace.	`AUTOSAR C++14 Rule A2-10-4`

AUTOSAR C++14 Rule	Description	Polyspace Checker
A2-10-5	An identifier name of a function with static storage duration or a non-member object with external or internal linkage should not be reused.	`AUTOSAR C++14 Rule A2-10-5`
A3-1-5	A function definition shall only be placed in a class definition if (1) the function is intended to be inlined (2) it is a member function template (3) it is a member function of a class template.	`AUTOSAR C++14 Rule A3-1-5`
A3-1-6	Trivial accessor and mutator functions should be inlined.	`AUTOSAR C++14 Rule A3-1-6`
A3-8-1	An object shall not be accessed outside of its lifetime.	`AUTOSAR C++14 Rule A3-8-1`
A5-1-6	Return type of a non-void return type lambda expression should be explicitly specified.	`AUTOSAR C++14 Rule A5-1-6`
A5-1-8	Lambda expressions should not be defined inside another lambda expression.	`AUTOSAR C++14 Rule A5-1-8`
A5-1-9	Identical unnamed lambda expressions shall be replaced with a named function or a named lambda expression.	`AUTOSAR C++14 Rule A5-1-9`
A5-2-1	dynamic_cast should not be used.	`AUTOSAR C++14 Rule A5-2-1`
A5-3-1	Evaluation of the operand to the typeid operator shall not contain side effects.	`AUTOSAR C++14 Rule A5-3-1`
A5-3-2	Null pointers shall not be dereferenced.	`AUTOSAR C++14 Rule A5-3-2`
A5-10-1	A pointer to member virtual function shall only be tested for equality with null-pointer-constant.	`AUTOSAR C++14 Rule A5-10-1`
A6-2-1	Move and copy assignment operators shall either move or respectively copy base classes and data members of a class, without any side effects.	`AUTOSAR C++14 Rule A6-2-1`
A6-2-2	Expression statements shall not be explicit calls to constructors of temporary objects only.	`AUTOSAR C++14 Rule A6-2-2`

AUTOSAR C++14 Rule	Description	Polyspace Checker
A6-5-3	Do statements should not be used.	`AUTOSAR C++14 Rule A6-5-3`
A7-1-1	Constexpr or const specifiers shall be used for immutable data declaration.	`AUTOSAR C++14 Rule A7-1-1`
A7-1-2	The constexpr specifier shall be used for values that can be determined at compile time.	`AUTOSAR C++14 Rule A7-1-2`
A7-1-5	The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax.	`AUTOSAR C++14 Rule A7-1-5`
A7-6-1	Functions declared with the [[noreturn]] attribute shall not return.	`AUTOSAR C++14 Rule A7-6-1`
A8-4-4	Multiple output values from a function should be returned as a struct or tuple.	`AUTOSAR C++14 Rule A8-4-4`
A8-4-14	Interfaces shall be precisely and strongly typed.	`AUTOSAR C++14 Rule A8-4-14`
A11-0-1	A non-POD type should be defined as class.	`AUTOSAR C++14 Rule A11-0-1`
A12-0-2	Bitwise operations and operations that assume data representation in memory shall not be performed on objects.	`AUTOSAR C++14 Rule A12-0-2`
A12-1-2	Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.	`AUTOSAR C++14 Rule A12-1-2`
A12-1-6	Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors.	`AUTOSAR C++14 Rule A12-1-6`
A12-4-2	If a public destructor of a class is non-virtual, then the class should be declared final.	`AUTOSAR C++14 Rule A12-4-2`

AUTOSAR C++14 Rule	Description	Polyspace Checker
A12-8-4	Move constructor shall not initialize its class members and base classes using copy semantics.	`AUTOSAR C++14 Rule A12-8-4`
A12-8-7	Assignment operators should be declared with the ref-qualifier &.	`AUTOSAR C++14 Rule A12-8-7`
A13-5-5	Comparison operators shall be non-member functions with identical parameter types and noexcept.	`AUTOSAR C++14 Rule A13-5-5`
A14-5-2	Class members that are not dependent on template class parameters should be defined in a separate base class.	`AUTOSAR C++14 Rule A14-5-2`
A14-5-3	A non-member generic operator shall only be declared in a namespace that does not contain class (struct) type, enum type or union type declarations.	`AUTOSAR C++14 Rule A14-5-3`
A15-1-1	Only instances of types derived from std::exception should be thrown.	`AUTOSAR C++14 Rule A15-1-1`
A15-1-3	All thrown exceptions should be unique.	`AUTOSAR C++14 Rule A15-1-3`
A15-2-1	Constructors that are not noexcept shall not be invoked before program startup.	`AUTOSAR C++14 Rule A15-2-1`
A15-3-3	Main function and a task main function shall catch at least: base class exceptions from all third-party libraries used, std::exception and all otherwise unhandled exceptions.	`AUTOSAR C++14 Rule A15-3-3`
A15-3-4	Catch-all (ellipsis and std::exception) handlers shall be used only in (a) main, (b) task main functions, (c) in functions that are supposed to isolate independent components and (d) when calling third-party code that uses exceptions not according to AUTOSAR C++14 guidelines.	`AUTOSAR C++14 Rule A15-3-4`

AUTOSAR C++14 Rule	Description	Polyspace Checker
A15-4-2	If a function is declared to be noexcept, noexcept(true) or noexcept(<true condition>), then it shall not exit with an exception.	`AUTOSAR C++14 Rule A15-4-2`
A15-4-3	Function's noexcept specification shall be either identical or more restrictive across all translation units and all overriders.	`AUTOSAR C++14 Rule A15-4-3`
A15-5-1	All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A noexcept exception specification shall be added to these functions as appropriate.	`AUTOSAR C++14 Rule A15-5-1`
A18-5-9	Custom implementations of dynamic memory allocation and deallocation functions shall meet the semantic requirements specified in the corresponding "Required behaviour" clause from the C++ Standard.	`AUTOSAR C++14 Rule A18-5-9`
A18-5-10	Placement new shall be used only with properly aligned pointers to sufficient storage capacity.	`AUTOSAR C++14 Rule A18-5-10`
A18-5-11	"operator new" and "operator delete" shall be defined together.	`AUTOSAR C++14 Rule A18-5-11`
A18-9-2	Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference.	`AUTOSAR C++14 Rule A18-9-2`
A18-9-4	An argument to std::forward shall not be subsequently used.	`AUTOSAR C++14 Rule A18-9-4`
A20-8-2	A std::unique_ptr shall be used to represent exclusive ownership.	`AUTOSAR C++14 Rule A20-8-2`
A20-8-3	A std::shared_ptr shall be used to represent shared ownership.	`AUTOSAR C++14 Rule A20-8-3`

AUTOSAR C++14 Rule	Description	Polyspace Checker
A20-8-5	std::make_unique shall be used to construct objects owned by std::unique_ptr.	`AUTOSAR C++14 Rule A20-8-5`
A20-8-6	std::make_shared shall be used to construct objects owned by std::shared_ptr.	`AUTOSAR C++14 Rule A20-8-6`
A26-5-2	Random number engines shall not be default-initialized.	`AUTOSAR C++14 Rule A26-5-2`
A27-0-2	A C-style string shall guarantee sufficient space for data and the null terminator.	`AUTOSAR C++14 Rule A27-0-2`
A27-0-3	Alternate input and output operations on a file stream shall not be used without an intervening flush or positioning call.	`AUTOSAR C++14 Rule A27-0-3`
M0-1-4	A project shall not contain non-volatile POD variables having only one use.	`AUTOSAR C++14 Rule M0-1-4`
M0-3-2	If a function generates error information, then that error information shall be tested.	`AUTOSAR C++14 Rule M0-3-2`
M7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	`AUTOSAR C++14 Rule M7-5-2`
M9-6-4	Named bit-fields with signed integer type shall have a length of more than one bit.	`AUTOSAR C++14 Rule M9-6-4`
M15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	`AUTOSAR C++14 Rule M15-1-1`
M15-3-1	Exceptions shall be raised only after start-up and before termination.	`AUTOSAR C++14 Rule M15-3-1`
M15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	`AUTOSAR C++14 Rule M15-3-4`

See also AUTOSAR C++14 Rules (Polyspace Bug Finder Access).

CERT C Support: Check for missing const-qualification and use of hardcoded numbers

Summary: In R2020b, you can look for violations of these CERT C recommendations in addition to previously supported rules.

CERT C Rule	Description	Polyspace Checker
DCL00-C	Const-qualify immutable objects	`CERT C: Rec. DCL00-C`

See also CERT C Rules and Recommendations (Polyspace Bug Finder Access).

CERT C++ Support: Check for exception handling issues, memory management problems, and other rule violations

Summary: In R2020b, you can look for violations of these CERT C++ rules in addition to previously supported rules.

CERT C++ Rule	Description	Polyspace Checker
ERR58-CPP	Handle all exceptions thrown before main() begins executing	`CERT C++: ERR58-CPP`
MEM54-CPP	Provide placement new with properly aligned pointers to sufficient storage capacity	`CERT C++: MEM54-CPP`
MEM55-CPP	Honor replacement dynamic storage management requirements	`CERT C++: MEM55-CPP`
MSC53-CPP	Do not return from a function declared [[noreturn]]	`CERT C++: MSC53-CPP`
ERR55-CPP	Honor exception specifications	`CERT C++: ERR55-CPP`

See also CERT C++ Rules (Polyspace Bug Finder Access).

MISRA C++:2008 Support: Check for commented out code, variables used once, exception handling issues, and other rule violations

Summary: In R2020b, you can look for violations of these MISRA C++:2008 rules in addition to previously supported rules.

MISRA C++:2008 Rule	Description	Polyspace Checker
0-1-4	A project shall not contain non-volatile POD variables having only one use.	`MISRA C++:2008 Rule 0-1-4`
0-3-2	If a function generates error information, then that error information shall be tested.	`MISRA C++:2008 Rule 0-3-2`

MISRA C++:2008 Rule	Description	Polyspace Checker
2-7-2	Sections of code should not be "commented out" using C-style comments.	`MISRA C++:2008 Rule 2-7-2`
2-7-3	Sections of code should not be "commented out" using C++-style comments.	`MISRA C++:2008 Rule 2-7-3`
14-5-1	A non-member generic function shall only be declared in a namespace that is not an associated namespace.	`MISRA C++:2008 Rule 14-5-1`
15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown	`MISRA C++:2008 Rule 15-1-1`
15-3-1	Exceptions shall be raised only after start-up and before termination of the program.	`MISRA C++:2008 Rule 15-3-1`
15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	`MISRA C++:2008 Rule 15-3-4`

See also MISRA C++:2008 Rules (Polyspace Bug Finder Access).

JSF AV C++ Support: Check for commented out code and methods that can be inlined

Summary: In R2020b, you can check for these JSF AV C++ rules in addition to previously supported rules.

Rule	Description
122	Trivial accessor and mutator functions should be inlined.
127	Code that is not used (commented out) shall be deleted.

See also JSF AV C++ Coding Rules (Polyspace Bug Finder Access).

MISRA C Support: Check for commented out code

Summary: In R2020b, you can look for violations of these MISRA C rules and directives in addition to previously supported rules and directives.

MISRA C Rule	Description	Polyspace Checker
MISRA C:2004 Rule 2.4	Sections of code should not be "commented out".	MISRA C:2004 Rule 2.4 See also MISRA C :2004 and MISRA AC AGC Coding Rules (Polyspace Bug Finder Access).
MISRA C:2012 Dir 4.4	Sections of code should not be "commented out".	`MISRA C:2012 Dir 4.4`

See also MISRA C :2012 Directives and Rules (Polyspace Bug Finder Access).

New Bug Finder Defect Checkers: Check for post-C++11 defects such as problematic move operations, missing constexpr, and noexcept violations

Summary: In R2020b, you can check for these new types of defects.

Defect	Description
`A move operation may throw`	Throwing move operations might result in STL containers using the corresponding copy operations
`Const std::move input may cause a more expensive object copy`	Const `std::move` input cannot be moved and results in more expensive copy operation
`Data race on adjacent bit fields`	Multiple threads perform unprotected operations on adjacent bit fields of a shared data structure
`Expensive std::string::c_str() use in a std::string operation`	An `std::string` operation uses the output of an `std::string::c_str` method, resulting in inefficient code
`Expensive constant std::string construction`	A string with unmodified content is reconstructed in different function calls or scopes, resulting in inefficient code
`Expensive copy in a range-based for loop iteration`	The loop variable of a range-based `for` loop copies the range elements instead of referencing them, resulting in inefficient code
`Expensive pass by value`	Functions pass large parameters by value instead of by reference
`Expensive return by value`	Functions return large parameters by value instead of by reference
`Incorrect value forwarding`	Forwarded object might be modified unexpectedly
`Missing constexpr specifier`	`constexpr` specifier can be used on expression for compile-time evaluation

Defect	Description
`Noexcept function exits with exception`	Functions specified as `noexcept`, `noexcept(true)` or `noexcept(<true condition>)` exit with an exception, which causes abnormal termination of program execution, leading to resource leak and security vulnerability
`std::move called on an unmovable type`	Result of `std::move` is not movable
`Throw argument raises unexpected exception`	The argument expression in a `throw` statement raises unexpected exceptions, leading to resource leaks and security vulnerabilities

See the full list of defect checkers in Defects (Polyspace Bug Finder Access).

Modifying Checker Behavior: Check for non-initialized buffers when passed by pointer to certain functions

Summary: In R2020b, you can indicate that pointer arguments to some functions must point to initialized buffers. By default, the checker `Non-initialized variable` checks a pointer for an initialized buffer only when you dereference the pointer. A function call such as:

```
int var; func(&var);
```

is not flagged for non-initialization because you might initialize the variable `var` in `func`. Starting in R2020b, you can specify a list of functions whose pointer arguments must be checked for initialized buffers.

For more information, see:

- `-code-behavior-specifications`
- Extend Checkers for Initialization to Check Function Arguments Passed by Pointers (Polyspace Bug Finder Server)

Benefits: Suppose that you consider some function calls as part of the system boundary and you want to make sure that you pass initialized buffers across the boundary. For instance, the Run-Time environment or `Rte_` functions in AUTOSAR allow a software component to communicate with other software components. You might want to ensure that pointer arguments to these functions point to initialized buffers. You can now use Bug Finder to find uninitialized buffers passed through pointers to these functions.

Changes in analysis options and binaries

XML syntax with option -code-behavior-specifications changed
Warns

The option `-code-behavior-specifications` takes an XML file as argument. You can use this XML file to specify whether a certain function must be subjected to special checks. For instance, you can specify that a function must not be used altogether.

In R2020b, the XML syntax changed slightly. To associate the behavior `FORBIDDEN_FUNC` with a function *`funcName`*, instead of the syntax:

```
<function name="funcName" behavior="FORBIDDEN_FUNC">
```

Use the syntax:

```
<function name="funcName">
    <behavior name="FORBIDDEN_FUNC">
</function>
```

See also `-code-behavior-specifications`.

Changes to coding rules checking

Summary: In R2020b, coding rules checking has improved across various coding standards:

- The Polyspace checkers for AUTOSAR C++14 now follow AUTOSAR C++14 release 18-10 (October 2018).
- You can check for MISRA® C++ and JSF AV C++ rules in the same run. If the issues that you want to detect span MISRA C++ and JSF AV C++, you can enable rules from both standards and detect all issues in a single run.

In addition, these changes have been made in checking of previously supported rules.

Rule	Description	Change
`MISRA C:2012 Dir 4.14`	The validity of values received from external sources shall be checked.	The checker now use a broader definition of valid data. The following are no longer considered as invalid data: • Inputs to functions that do not have a visible caller • Return values of undefined (stubbed) functions • Global variables external to the unit See Sources of Tainting in a Polyspace Analysis. To revert to the previous definition, use the option `-consider-analysis-perimeter-as-trust-boundary`.

Rule	Description	Change
`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.	The checker takes into account header files irrespective of whether you suppress headers using the option `Do not generate results for (-do-not-generate-results-for)`. For instance, the checker raises a violation if the number of macros in C99 code exceeds 4095. The checker now counts macros in header files irrespective of whether you choose to suppress results in headers. The reason is that the header files are included in a translation unit and the translation unit as a whole is subject to MISRA C: 2012 Rule 1.1. Previously, the headers were taken into account only if unsuppressed.
`AUTOSAR C++14 Rule A2-13-6`	Universal character names shall be used only inside character or string literals.	The checker no longer flags universal character names in code deactivated with a preprocessor directive such as `#if`. You can enter universal character names for non-string uses in deactivated code.
`AUTOSAR C++14 Rule A5-1-1`	Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead.	The checker now flags use of literal values as template parameters.
`MISRA C++:2008 Rule 2-10-2`	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.	The checker no longer flags class member operators in nested scopes. Class member operators in nested scopes do not hide each other.
`MISRA C++:2008 Rule 3-4-1`	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.	The checker no longer flags identifiers used only in a range-based `for` loop but defined outside the loop.

Rule	Description	Change
AUTOSAR C++14 Rule A21-8-1	Arguments to character-handling functions shall be representable as an unsigned char.	The checker now only detects the use of a signed or plain `char` variable with a negative value as argument to a character-handling function declared in `<cctype>`, for instance, `isalpha()` or `isdigit()`.
MISRA C++:2008 Rule 14-6-2	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.	The checker no longer flags calls that use an underlying function call operator.
MISRA C++:2008 Rule 17-0-1	Reserved identifiers, macros and functions in the Standard Library shall not be defined, redefined or undefined.	The checker raises a violation if you define or redefine a macro beginning with an underscore followed by an uppercase letter. These macros are typically reserved for the Standard Library.
CERT C: Rec. PRE01-C	Use parentheses within macros around parameter names.	The checker no longer flags uses of the `va_arg` macro if the macro parameters are not enclosed in parentheses (in accordance with the exception in the CERT C specifications).
CERT C: Rule MSC39-C	Do not call va_arg() on a va_list that has an indeterminate value.	The checker does not flag an uninitialized `va_list` variable if the variable is only passed to another function (unless the function belongs to the standard `vprintf`-family).
CERT C++: DCL51-CPP	Do not declare or define a reserved identifier.	The checker now flags: • Macros or identifiers beginning with underscore followed by an uppercase letter. • User-defined literal operators if the operator names do not begin with an underscore (C++11 and later). By convention, these macros, identifiers and operators are reserved for the Standard Library.

Rule	Description	Change
`CERT C++: EXP52-CPP`	Do not rely on side effects in unevaluated operands.	The checker now flags `decltype` operations where the operands have side effects.
`CERT C: Rule EXP36-C` and `CERT C++: EXP36-C`	Do not cast pointers into more strictly aligned pointer types.	The checker now flags: • Conversion of `void*` pointer into pointer to object. • Source buffer misaligned with destination buffer.
`CERT C: Rule MSC39-C` and `CERT C++: MSC39-C`	Do not call va_arg() on a va_list that has an indeterminate value.	The checker flags situations where you might be using a `va_list` that has an indeterminate value.
`CERT C: Rule MEM30-C` and `CERT C++: MEM30-C`	Do not access freed memory.	This checker now flags attempts to deallocate a previously freed memory block.
`CERT C: Rule MEM35-C` and `CERT C++: MEM35-C`	Allocate sufficient memory for an object.	This checker now flags the use of a pointer type as the argument of the `sizeof` operator in a `malloc` statement. Use the type of the object to which the pointer points as the argument of the `sizeof` operator.
`CERT C: Rule EXP43-C`	Avoid undefined behavior when using restrict-qualified pointers.	The checker now detects situations where you assign a `restrict` qualified pointer to another `restrict` qualified pointer such that they both attempt to point to the same object.
`CERT C: Rule EXP46-C` and `CERT C++: EXP46-C`	Do not use a bitwise operator with a Boolean-like operand.	This checker now flags the use of bitwise operators, such as: • Bitwise AND (&, &=) • Bitwise OR (`|`, `|=`) • Bitwise XOR (^, ^=) • Bitwise NOT(~) with: • Boolean type variables • Outputs of relational or equality expressions

Rule	Description	Change
`CERT C: Rule STR37-C` and `CERT C++: STR37-C`	Arguments to character-handling functions must be representable as an unsigned char.	The checker now only detects the use of a signed or plain `char` variable with a negative value as argument to a character-handling function declared in `ctype.h`, for instance, `isalpha()` or `isdigit()`.

Rule	Description	Change
Coding rules that involve detection of tainted data, including: • `CERT C: Rec. INT04-C` • `CERT C: Rec. INT10-C` • `CERT C: Rule INT31-C` and `CERT C++: INT31-C` • `CERT C: Rule INT32-C` and `CERT C++: INT32-C` • `CERT C: Rule INT33-C` and `CERT C++: INT33-C` • `CERT C: Rule ARR30-C` and `CERT C++: ARR30-C` • `CERT C: Rule ARR32-C` • `CERT C: Rule ARR38-C` and `CERT C++: ARR38-C` • `CERT C: Rec. STR02-C` • `CERT C: Rule STR32-C` and `CERT C++: STR32-C` • `CERT C: Rec. MEM04-C` • `CERT C: Rec. MEM05-C` • `CERT C: Rule MEM35-C` and `CERT C++: MEM35-C` • `CERT C: Rule FIO30-C` and `CERT C++: FIO30-C` • `CERT C: Rec. ENV01-C` • `CERT C: Rec. MSC21-C` • `CERT C: Rec. WIN00-C` • `AUTOSAR C++14 Rule A5-6-1` • `ISO/IEC TS 17961 [usrfmt]` • `ISO/IEC TS 17961 [taintstrcpy]` • `ISO/IEC TS 17961 [taintformatio]` • `ISO/IEC TS 17961 [taintsink]`		The checkers now use a narrower definition of tainted data. The following are no longer considered as tainted data: • Inputs to functions that do not have a visible caller • Return values of undefined (stubbed) functions • Global variables external to the unit See Sources of Tainting in a Polyspace Analysis. To revert to the previous definition, use the option `-consider-analysis-perimeter-as-trust-boundary`.

Rule	Description	Change
`MISRA C++:2008 Rule 0-1-4` and `AUTOSAR C++14 Rule M0-1-4`	A project shall not contain non-volatile POD variables having only one use.	• The checker now considers dynamic assignments of a variable, such as `int var = foo()` as a single use of the variable. • Some objects are designed to be used only once by their semantics. Polyspace does not flag a single use of these objects: • `lock_guard` • `scoped_lock` • `shared_lock` • `unique_lock` • `thread` • `future` • `shared_future` If you use nonstandard objects that provide similar functionality as the objects in the preceding list, Polyspace might flag single uses of the nonstandard objects. Justify their single uses by using comments.

Compatibility Considerations

If you checked your code for the preceding rules, you might see a change in the number of violations.

Updated Bug Finder defect checkers

Summary: In R2020b, these defect checkers have been updated.

Defect	Description	Update
`Deterministic random output from constant seed` and `Predictable random output from predictable seed`	Issues with seeding of random number generator functions	The checkers now support random number generator functions from the C++ Standard Library, for instance, `std::linear_congruential_engine<>::seed()` and `std::mersenne_twister_engine<>::seed()`.

Defect	Description	Update
Tainted Data Defects (Polyspace Bug Finder)	Use of tainted and unvalidated data in critical operations	The checkers now use a narrower definition of tainted data. The following are no longer considered as tainted data: • Inputs to functions that do not have a visible caller • Return values of undefined (stubbed) functions • Global variables external to the unit See Sources of Tainting in a Polyspace Analysis. To revert to the previous definition, use the option `-consider-analysis-perimeter-as-trust-boundary`.
Large pass-by-value argument	Functions pass large parameters by value instead of by reference	Checker is removed. Use `Expensive pass by value` and `Expensive return by value` instead.
• `Empty destructors may cause unnecessary data copies` • `std::endl may cause an unnecessary flush`	Issues that impact performance of C++ code	The **Impact** attribute of these checkers have been changed from `High` to `Low`. These checkers do not have a universally high criticality. The checkers are critical only for code that must be optimized for performance.
`Inefficient string length computation`	Issue that impacts performance of C++ code	The **Impact** attribute of this checker has been changed from `High` to `Medium`. This checker does not have a universally high criticality. The checker is critical only for code that must be optimized for performance and also promotes a good coding style.
`Missing return statement`	Issues with data flow	This checker flags nonvoid functions that do not return the flow of execution except if the function is specified as `[[noreturn]]`.

Compatibility Considerations

If you check your code for the preceding defects, you might see a difference in the number of issues found.

Updated code metrics specifications

Summary: In R2020b, these code metrics specifications have been updated.

Code Metric	Update

`Number of Called Functions`	These metrics now accounts for function calls in a C++ constructor initializer list.
	
	For instance, in this code snippet, the number of called functions of `Derived::Derived()` is one. Previously, the number was computed as zero.
	
	`class  Base`
	`{`
	`  int b;`
	`  public:`
	`      Base() {`
	`          b = 0;`
	`      };`
	`};`
	`class Derived : public Base`
	`{`
	`  int d;`
	`  public:`
	`      Derived() : Base() {`
	`          d = 0;`
	`      };`
	`};`

Compatibility Considerations

If you compute these code metrics, you can see a difference in results compared to previous releases.

R2020a

Compiler Support: Set up Polyspace analysis easily for code compiled with MPLAB XC8 C compilers

Summary: If you build your source code by using MPLAB XC8 C compilers, in R2020a, you can specify the compiler name for your Polyspace analysis.

You specify a compiler using the option `Compiler (-compiler)`.

```
polyspace-bug-finder-server -compiler microchip -target pic -sources file.c ....
```

See also `MPLAB XC8 C Compiler (-compiler microchip)`.

Benefits: You can now set up a Polyspace project without knowing the internal workings of MPLAB XC8 C compilers. If your code compiles with your compiler, it will compile with Polyspace in most cases without requiring additional setup. Previously, you had to explicitly define macros that were implicitly defined by the compiler and remove unknown language extensions from your preprocessed code.

Compiler Support: Set up Polyspace analysis to emulate MPLAB XC16 and XC32 compilers

Summary: If you use MPLAB XC16 or XC32 compilers to build your source code, in R2020a, you can easily emulate these compilers by using the Polyspace GCC compiler options. See Emulate Microchip MPLAB XC16 and XC32 Compilers.

For each compiler, you can emulate these target processor types:

- **MPLAB XC16**: Targets PIC24 and dsPIC.
- **MPLAB XC32**: Target PIC32.

Benefits: You can copy the analysis options required for emulating MPLAB XC16 or XC32 compilers and paste into your Polyspace options file (or specify in a Polyspace project in the user interface), and avoid compilation errors from issues specific to these compilers.

Source Code Encoding: Non-ASCII characters in source code analyzed and displayed without errors

Summary: In R2020a, if your source code contains non-ASCII characters, for instance, Japanese or Korean characters, the Polyspace analysis can interpret the characters and later display the source code correctly.

If you still have compilation errors or display issues from non-ASCII characters, you can explicitly specify your source code encoding using the option `Source code encoding (-sources-encoding)`.

Extending Checkers: Run stricter analysis that considers all possible values of system inputs

Summary: In R2020a, you can run a stricter Polyspace Bug Finder™ analysis that checks the robustness of your code against numerical edge cases. For defects that are detected with the stricter checks, the analysis can also show an example of values that lead to the defect. Use the option `Run` `stricter checks considering all values of system inputs (-checks-using-system-input-values)` to enable the stricter checks.

Benefits: For a subset of **Numerical** and **Static memory** defect checkers, the analysis considers all possible values of:

- Global variables
- Reads of volatile variables
- Returns of stubbed functions
- Inputs to the functions you specify with the option `Consider inputs to these functions (-system-inputs-from)`

See also Extend Bug Finder Checkers to Find Defects from Specific System Input Values.

AUTOSAR C++14 Support: Check for 37 new rules related to lexical conventions, standard conversions, declarations, derived classes, special member functions, overloading and other groups

Summary: In R2020a, you can look for violations of these AUTOSAR C++14 rules in addition to previously supported rules.

AUTOSAR C++14 Rule	Description	Polyspace Checker
A0-1-5	There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it.	`AUTOSAR C++14 Rule A0-1-5`
A2-3-1	Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code.	`AUTOSAR C++14 Rule A2-3-1`
A2-7-1	The character \ shall not occur as a last character of a C++ comment.	`AUTOSAR C++14 Rule A2-7-1`
A2-10-1	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.	`AUTOSAR C++14 Rule A2-10-1`
A2-10-6	A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope.	`AUTOSAR C++14 Rule A2-10-6`
A2-13-4	String literals shall not be assigned to non-constant pointers.	`AUTOSAR C++14 Rule A2-13-4`
A2-13-6	Universal character names shall be used only inside character or string literals.	`AUTOSAR C++14 Rule A2-13-6`

AUTOSAR C++14 Rule	Description	Polyspace Checker
A3-3-2	Static and thread-local objects shall be constant-initialized.	`AUTOSAR C++14 Rule A3-3-2`
A4-5-1	Expressions with type enum or enum class shall not be used as operands to built-in and overloaded operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.	`AUTOSAR C++14 Rule A4-5-1`
A4-10-1	Only nullptr literal shall be used as the null-pointer-constraint.	`AUTOSAR C++14 Rule A4-10-1`
A7-1-3	CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name.	`AUTOSAR C++14 Rule A7-1-3`
A7-1-8	A non-type specifier shall be placed before a type specifier in a declaration.	`AUTOSAR C++14 Rule A7-1-8`
A7-4-1	The asm declaration shall not be used.	`AUTOSAR C++14 Rule A7-4-1`
A8-2-1	When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters.	`AUTOSAR C++14 Rule A8-2-1`
A8-5-3	A variable of type auto shall not be initialized using {} or ={} braced-initialization.	`AUTOSAR C++14 Rule A8-5-3`
A10-1-1	Class shall not be derived from more than one base class which is not an interface class.	`AUTOSAR C++14 Rule A10-1-1`
A10-3-1	Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final.	`AUTOSAR C++14 Rule A10-3-1`
A10-3-2	Each overriding virtual function shall be declared with the override or final specifier.	`AUTOSAR C++14 Rule A10-3-2`
A10-3-3	Virtual functions shall not be introduced in a final class.	`AUTOSAR C++14 Rule A10-3-3`
A10-3-5	A user-defined assignment operator shall not be virtual.	`AUTOSAR C++14 Rule A10-3-5`

AUTOSAR C++14 Rule	Description	Polyspace Checker
A11-0-2	A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class.	`AUTOSAR C++14 Rule A11-0-2`
A12-0-1	If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well.	`AUTOSAR C++14 Rule A12-0-1`
A12-4-1	Destructor of a base class shall be public virtual, public override or protected non-virtual.	`AUTOSAR C++14 Rule A12-4-1`
A12-8-6	Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class.	`AUTOSAR C++14 Rule A12-8-6`
A13-1-2	User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters.	`AUTOSAR C++14 Rule A13-1-2`
A13-2-3	A relational operator shall return a boolean value.	`AUTOSAR C++14 Rule A13-2-3`
A13-5-1	If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented.	`AUTOSAR C++14 Rule A13-5-1`
A13-5-2	All user-defined conversion operators shall be defined explicit.	`AUTOSAR C++14 Rule A13-5-2`
A14-7-2	Template specialization shall be declared in the same file (1) as the primary template (2) as a user-defined type, for which the specialization is declared.	`AUTOSAR C++14 Rule A14-7-2`
A14-8-2	Explicit specializations of function templates shall not be used.	`AUTOSAR C++14 Rule A14-8-2`

AUTOSAR C++14 Rule	Description	Polyspace Checker
A16-6-1	#error directive shall not be used.	`AUTOSAR C++14 Rule A16-6-1`
A17-6-1	Non-standard entities shall not be added to standard namespaces.	`AUTOSAR C++14 Rule A17-6-1`
A18-1-3	The std::auto_ptr shall not be used.	`AUTOSAR C++14 Rule A18-1-3`
A18-1-6	All std::hash specializations for user-defined types shall have a noexcept function call operator.	`AUTOSAR C++14 Rule A18-1-6`
A18-5-2	Operators new and delete shall not be called explicitly.	`AUTOSAR C++14 Rule A18-5-2`
A18-9-3	The std::move shall not be used on objects declared const or const&.	`AUTOSAR C++14 Rule A18-9-3`
A23-0-1	An iterator shall not be implicitly converted to const_iterator.	`AUTOSAR C++14 Rule A23-0-1`

CERT C Support: Check for CERT C rules related to threads and hardcoded sensitive data, and recommendations related to macros and code formatting

Summary: In R2020a, you can look for violations of these CERT C rules and recommendations in addition to the previously supported ones. With these new rules, all CERT C rules can be checked with Bug Finder.

Rules

CERT C Rule	Description	Polyspace Checker
CON34-C	Declare objects shared between threads with appropriate storage durations	`CERT C: Rule CON34-C`
CON38-C	Preserve thread safety and liveness when using condition variables	`CERT C: Rule CON38-C`
MSC41-C	Never hard code sensitive information	`CERT C: Rule MSC41-C`
POS47-C	Do not use threads that can be canceled asynchronously	`CERT C: Rule POS47-C`
POS50-C	Declare objects shared between POSIX threads with appropriate storage durations	`CERT C: Rule POS50-C`
POS53-C	Do not use more than one mutex for concurrent waiting operations on a condition variable	`CERT C: Rule POS53-C`

Recommendations

CERT C Recommendation	Description	Polyspace Checker
PRE10-C	Wrap multistatement macros in a do-while loop	`CERT C: Rec. PRE10-C`
PRE11-C	Do not conclude macro definitions with a semicolon	`CERT C: Rec. PRE11-C`
EXP15-C	Do not place a semicolon on the same line as an if, for, or while statement	`CERT C: Rec. EXP15-C`

CERT C++ Support: Check for CERT C++ rule related to order of initialization in constructor

Summary: In R2020a, you can look for violations of these CERT C++ rules in addition to previously supported rules.

CERT C++ Rule	Description	Polyspace Checker
DCL58-CPP	Do not modify the standard namespaces	`CERT C++: DCL58-CPP`
MSC41-C	Never hard code sensitive information	`CERT C++: MSC41-C`
OOP53-CPP	Write constructor member initializers in the canonical order	`CERT C++: OOP53-CPP`

CWE Support: Check for CWE rule related to incorrect block delimitation

Summary: In R2020a, you can check for violation of this CWE rule in addition to previously supported rules.

CWE Rule	Description	Polyspace Checkers
483	Incorrect block delimitation	• `Incorrectly indented statement` • `Semicolon on same line as if, for or while statement`

For the full mapping between CWE rules and Polyspace Bug Finder defect checkers, see CWE Coding Standard and Polyspace Results.

New Bug Finder Defect Checkers: Check for possible performance bottlenecks, hardcoded sensitive data and other issues

Summary: In R2020a, you can check for new types of defects.

A new category of C++-specific checkers checks for constructs that might cause performance issues and suggests more efficient alternatives. Other checkers include security checkers for hard coded sensitive data, good practice checkers for issues such as ill-formed macros and concurrency checkers for issues such as asynchronously cancellable threads.

Performance Checkers

Defect	Description
`Const parameter values may cause unnecessary data copies`	Const parameter values prevent a move operation resulting in a more performance-intensive copy operation
`Const return values may cause unnecessary data copies`	Const return values prevent a move operation resulting in a more performance-intensive copy operation
`Empty destructors may cause unnecessary data copies`	User-defined empty destructors prevent autogeneration of move constructors and move assignment operators
`Inefficient string length computation`	String length calculated by using string length functions on return from `std::basic_string::c_str()` instead of using `std::basic_string::length()`
`std::endl may cause an unnecessary flush`	`std::endl` is used instead of more efficient alternatives such as `\n`

Other Checkers

Defect	Description
`Asynchronously cancellable thread`	Calling thread might be cancelled in an unsafe state
`Automatic or thread local variable escaping from a thread`	Variable is passed from one thread to another without ensuring that variable stays alive for duration of both threads
`Hard-coded sensitive data`	Sensitive data is exposed in code, for instance as string literals
`Incorrectly indented statement`	Statement indentation incorrectly makes it appear as part of a block
`Macro terminated with a semicolon`	Macro definition ends with a semicolon
`Macro with multiple statements`	Macro consists of multiple semicolon-terminated statements, enclosed in braces or not
`Missing final step after hashing update operation`	Hash is incomplete or non-secure
`Missing private key for X.509 certificate`	Missing key might result in run-time error or non-secure encryption
`Move operation on const object`	`std::move` function is called with object declared `const` or `const&`
`Multiple mutexes used with same conditional variable`	Threads using different mutexes when concurrently waiting on the same condition variable is undefined behavior
`Multiple threads waiting on same condition variable`	Using `cnd_signal` to wake up one of the threads might result in indefinite blocking
`No data added into context`	Performing hash operation on empty context might cause run-time errors
`Possibly inappropriate data type for switch expression`	Switch expression has a data type other than char, short, int or enum
`Semicolon on the same line as an if, for or while statement`	Semicolon on same line results in empty body of if, for or while statement
`Server certificate common name not checked`	Attacker might use valid certificate to impersonate trusted host
`TLS/SSL connection method not set`	Program cannot determine whether to call client or server routines
`TLS/SSL connection method set incorrectly`	Program calls functions that do not match role set by connection method
`Unmodified variable not const-qualified`	Variable is not `const`-qualified but no modification anywhere in the program
`Use of a forbidden function`	Function appears in a blacklist of forbidden functions
`Redundant expression in sizeof operand`	`sizeof` operand contains expression that is not evaluated

Defect	Description
`X.509 peer certificate not checked`	Connection might be vulnerable to man-in-the-middle attacks

Modifying Checkers: Create list of functions to prohibit and check for use of functions from the list

Summary: In R2020a, you can define a blacklist of functions to forbid from your source code. The Bug Finder checker `Use of a forbidden function` checks if a function from this list appears in your sources.

Benefits: A function might be blacklisted for one of these reasons:

- The function can lead to many situations where the behavior is undefined leading to security vulnerabilities, and a more secure function exists.

  You can blacklist functions that are not explicitly checked by existing checkers such as `Use of dangerous standard function` or `Use of obsolete standard function`.
- The function is being deprecated as part of a migration, for instance, from C++98 to C++11.

  As part of a migration, you can make a list of functions that need to be replaced and use this checker to identify their use.

See also Flag Deprecated or Unsafe Functions Using Bug Finder Checkers.

Exporting Results: Export only results that must be reviewed to satisfy software quality objectives (SQOs)

Summary: In R2020a, when exporting Polyspace results from the Polyspace Access web interface to a text file, you can export only those results that must be fixed or justified to satisfy your software quality objectives. The software quality objectives are specified through a progressively stricter set of SQO levels, numbered from 1 to 6.

See also:

- `polyspace-access`
- Send Email Notifications with Polyspace Bug Finder Results
- Bug Finder Quality Objectives (Polyspace Bug Finder Access)

Benefits: You can customize the requirements of each level in the Polyspace Access web interface, and then use the option `-open-findings-for-sqo` with the level number to export only those results that must be reviewed to meet the requirements.

Jenkins Support: Use sample Jenkins Pipeline script to run Polyspace as part of continuous delivery pipeline

Summary: In R2020a, you can start from a template Jenkins Pipeline script to run Polyspace analysis as part of a continuous delivery pipeline.

See Sample Jenkins Pipeline Scripts for Polyspace Analysis.

Benefits: You can make simple replacements to adapt the template to your Polyspace Server and Access installations, and include the script in a new or existing Jenkinsfile to get up and running with Polyspace in Jenkins Pipelines.

Report Generation: Configure report generator to communicate with Polyspace Access over HTTPS

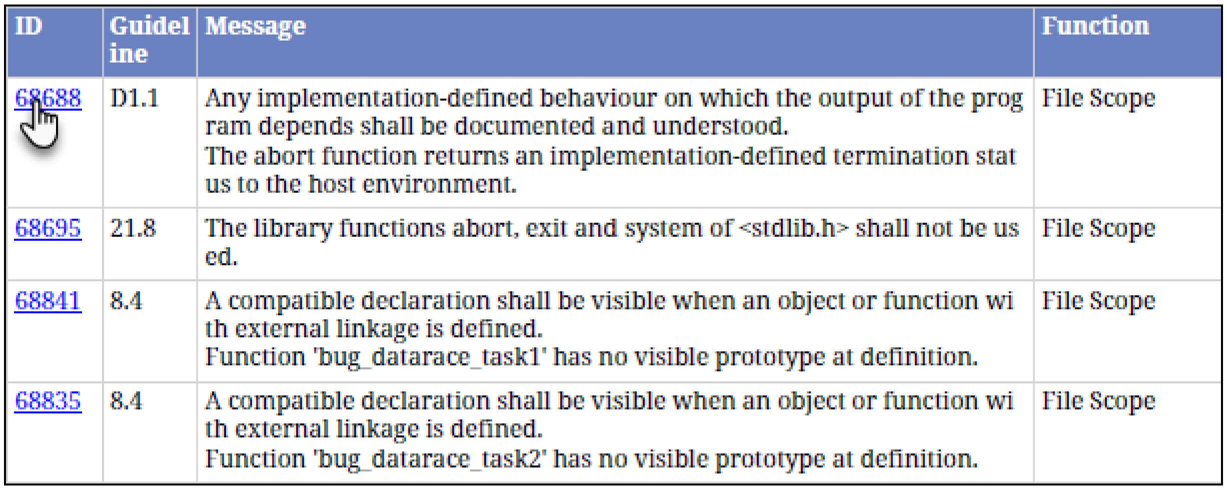
In R2020a, if you generate reports for results that are stored on Polyspace Access, you can configure the `polyspace-report-generator` binary to communicate with Polyspace Access over HTTPS.

Use the `-configure-keystore` option to run this one-time configuration step. See `polyspace-report-generator`.

Previously, you needed a Polyspace Bug Finder desktop license to generate reports if Polyspace Access was configured with HTTPS.

Report Generation: Navigate to Polyspace Access Results List from report

In R2020a, if you generate a report for results that are stored on Polyspace Access, you can navigate from the report to the **Results List** in the Polyspace Access web interface.



ID	Guideline	Message	Function
68688	D1.1	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood. The abort function returns an implementation-defined termination status to the host environment.	File Scope
68695	21.8	The library functions abort, exit and system of <stdlib.h> shall not be used.	File Scope
68841	8.4	A compatible declaration shall be visible when an object or function with external linkage is defined. Function 'bug_datarace_task1' has no visible prototype at definition.	File Scope
68835	8.4	A compatible declaration shall be visible when an object or function with external linkage is defined. Function 'bug_datarace_task2' has no visible prototype at definition.	File Scope

Click the link in the **ID** column to open Polyspace Access with the **Results List** filtered down to the corresponding finding.

Changes in analysis options and binaries

Option -function-behavior-specifications renamed to -code-behavior-specifications and capabilities extended

Warns

The option `-function-behavior-specifications` has been renamed to `-code-behavior-specifications`.

Using this option, you could previously map your functions to standard library functions to work around analysis imprecisions or specify thread creation routines. Now, you can use the option to define a blacklist of functions to forbid from your source code.

See also `-code-behavior-specifications`.

Changes to coding rules checking

Summary: In R2020a, the following changes have been made in checking of previously supported rules.

Rule	Description	Change
Some MISRA C: 2012 rules that were previously specific to a C standard	• C90-specific rules: `8.1`, `17.3` • C99-specific rules: `3.2`, `8.10`, `21.11`, `21.12`	These rules are now checked irrespective of the C standard. The reason is that the constructs flagged by these rules can be found in code using either standard, possibly with language extensions.
`MISRA C:2012 Rule 8.4`	A compatible declaration shall be visible when an object with an external linkage is defined.	• The checker now flags tentative definitions (variables declared without an `extern` specifier and not explicitly defined), for instance: `uint8_t var;` • The checker does not raise a violation on the `main` function.

`MISRA C++:2008 Rule 0-1-3`, `AUTOSAR C++14 Rule M0-1-3`	A project shall not contain unused variables.	The checker does not flag as unused constants used in template instantiations, such as the variable `size` here:
		`const std::uint8_t size = 2;`
		`std::array<uint8_t, size> arr = {0,1};`

`MISRA C++:2008 Rule 2-10-5`	The identifier name of a non-member object or function with static duration should not be reused.	The checker does not flag situations where a variable defined in a header file appears to be reused because the header file is included more than once, possibly along different inclusion paths.

Rule	Description	Change
`MISRA C++:2008 Rule 18-4-1`	Dynamic heap memory allocation shall not be used.	The checker now flags uses of the `alloca` function. Though memory leak cannot happen with the `alloca` function, other issues associated with dynamic memory allocation, such as memory exhaustion and nondeterministic behavior, can still occur.

Updated Bug Finder defect checkers

Summary: In R2020a, these defect checkers have been updated.

Defect	Description	Update
`Copy constructor not called in initialization list`	Copy constructor does not call copy constructors of some data members	The checker no longer flags copy constructors in templates. In template declarations, the member data types are not known and it is not clear which constructors need to be called.
`Dead code`	Code does not execute	If a `try` block contains a `return` statement, the checker no longer flags the corresponding `catch` block as dead code. A `return` statement involves a copy and copy constructors that are called might throw exceptions, resulting in the `catch` block being executed.
`Missing explicit keyword`	One-parameter constructor missing the `explicit` specifier	The checker has been updated to include user-defined conversion operators declared or defined in-class without the `explicit` keyword.
`Missing return statement`	Function does not return value though the return type is not void	The checker respects the option `-termination-functions`. If Bug Finder incorrectly flags a missing `return` statement on a path where a process termination function exists, you can make the analysis aware of the process termination function using this option.

Compatibility Considerations

If you check for the defects mentioned above, you can see a difference in the number of issues found.

R2019b

Compiler Support: Set up Polyspace analysis easily for code compiled with Cosmic compilers

Summary: If you build your source code by using Cosmic compilers, in R2019b, you can specify the compiler name for your Polyspace analysis.

You specify a compiler using the option `Compiler (-compiler)`.

```
polyspace-bug-finder-server -compiler cosmic -target s12z -sources file.c ....
```

Benefits: You can now set up a Polyspace project without knowing the internal workings of Cosmic compilers. If your code compiles with your compiler, it will compile with Polyspace in most cases without requiring additional setup. Previously, you had to explicitly define macros that were implicitly defined by the compiler and remove unknown language extensions from your preprocessed code.

AUTOSAR C++14 Support: Check for misuse of lambda expressions, potential problems with enumerations, and other issues

In R2019b, you can look for violations of these AUTOSAR C++14 rules in addition to previously supported rules.

AUTOSAR C++14 Rule	Description	Polyspace Checker
A0-1-4	There shall be no unused named parameters in non-virtual functions.	`AUTOSAR C++14 Rule A0-1-4`
A3-1-2	Header files, that are defined locally in the project, shall have a file name extension of one of: `.h`, `.hpp` or `.hxx`.	`AUTOSAR C++14 Rule A3-1-2`
A5-1-2	Variables shall not be implicitly captured in a lambda expression.	`AUTOSAR C++14 Rule A5-1-2`
A5-1-3	Parameter list (possibly empty) shall be included in every lambda expression.	`AUTOSAR C++14 Rule A5-1-3`
A5-1-4	A lambda expression shall not outlive any of its reference-captured objects.	`AUTOSAR C++14 Rule A5-1-4`
A5-1-7	A lambda shall not be an operand to `decltype` or `typeid`.	`AUTOSAR C++14 Rule A5-1-7`
A5-16-1	The ternary conditional operator shall not be used as a sub-expression.	`AUTOSAR C++14 Rule A5-16-1`
A7-2-2	Enumeration underlying base type shall be explicitly defined.	`AUTOSAR C++14 Rule A7-2-2`
A7-2-3	Enumerations shall be declared as scoped enum classes.	`AUTOSAR C++14 Rule A7-2-3`

AUTOSAR C++14 Rule	Description	Polyspace Checker
A16-0-1	The preprocessor shall only be used for unconditional and conditional file inclusion and include guards, and using the following directives: (1) `#ifndef`, (2) `#ifdef`, (3) `#if`, (4) `#if defined`, (5) `#elif`, (6) `#else`, (7) `#define`, (8) `#endif`, (9) `#include`	`AUTOSAR C++14 Rule A16-0-1`
A16-7-1	The `#pragma` directive shall not be used.	`AUTOSAR C++ 14 Rule A16-7-1`
A18-1-1	C-style arrays shall not be used.	`AUTOSAR C++ 14 Rule A18-1-1`
A18-1-2	The `std::vector<bool>` specialization shall not be used.	`AUTOSAR C++ 14 Rule A18-1-2`
A18-5-1	Functions `malloc`, `calloc`, `realloc` and `free` shall not be used.	`AUTOSAR C++ 14 Rule A18-5-1`
A18-9-1	The `std::bind` shall not be used.	`AUTOSAR C++ 14 Rule A18-9-1`

For all supported AUTOSAR C++14 rules, see AUTOSAR C++14 Rules (Polyspace Bug Finder Access).

CERT C++ Support: Check for pointer escape via lambda expressions, exceptions caught by value, use of bytewise operations for copying objects, and other issues

In R2019b, you can look for violations of these CERT C++ rules in addition to previously supported rules.

CERT C++ Rule	Description	Polyspace Checker
DCL59-CPP	Do not define an unnamed namespace in a header file	`CERT C++: DCL59-CPP`
EXP61-CPP	A lambda object shall not outlive any of its reference captured objects.	`CERT C++: EXP61-CPP`
MEM57-CPP	Avoid using default operator new for over-aligned types	`CERT C++: MEM57-CPP`
ERR61-CPP	Catch exceptions by lvalue reference	`CERT C++: ERR61-CPP`
OOP57-CPP	Prefer special member functions and overloaded operators	`CERT C++: OOP57-CPP`

For all supported CERT C++ rules, see CERT C++ Rules (Polyspace Bug Finder Access).

CERT C Support: Check for undefined behavior from successive joining or detaching of the same thread

In R2019b, you can look for violations of these CERT C rules in addition to previously supported rules.

CERT C Rule	Description	Polyspace Checker
CON39-C	Do not join or detach a thread that was previously joined or detached	`CERT C: Rule CON39-C`

For all supported CERT C guidelines, see CERT C Rules and Recommendations (Polyspace Bug Finder Access).

New Bug Finder Defect Checkers: Check for new security vulnerabilities, multithreading issues, missing C++ overloads, and other issues

Summary: In R2019b, you can check for the following new types of defects.

Defect	Description
`Unnamed namespace in header file`	Header file contains unnamed namespace leading to multiple definitions
`Lambda used as decltype or typeid operand`	`decltype` or `typeid` is used on lambda expression
`Operator new not overloaded for possibly overaligned class`	Allocated storage might be smaller than object alignment requirement
`Bytewise operations on nontrivial class object`	Value representations may be improperly initialized or compared
`Missing hash algorithm`	Context in EVP routine is initialized without a hash algorithm
`Missing salt for hashing operation`	Hashed data is vulnerable to rainbow table attack
`Missing X.509 certificate`	Server or client cannot be authenticated
`Missing certification authority list`	Certificate for authentication cannot be trusted
`Missing or double initialization of thread attribute`	Noninitialized thread attribute used in functions that expect initialized attributes or duplicated initialization of thread attributes
`Use of undefined thread ID`	Thread ID from failed thread creation used in subsequent thread functions
`Join or detach of a joined or detached thread`	Thread that was previously joined or detached is joined or detached again

MISRA C:2012 Directive 4.12: Dynamic memory allocation shall not be used

Summary: In R2019b, you can look for violations of MISRA C:2012 Directive 4.12. The directive states that dynamic memory allocation and deallocation packages provided by the Standard Library or third-party packages shall not be used. The use of these packages can lead to undefined behavior.

See `MISRA C:2012 Dir 4.12`.

Configuration from Build System: Compiler version automatically detected from build system

Summary: In R2019b, if you create a Polyspace analysis configuration from your build system by using the `polyspace-configure` command, the analysis uses the correct compiler version for the option `Compiler (-compiler)` for GNU® C, Clang, and Microsoft® Visual C++® compilers. You do not have to change the compiler version before starting the Polyspace analysis.

Benefits: Previously, if you traced your build system to create a Polyspace analysis configuration, the latest supported compiler version was used in the configuration. If your code was compiled with an earlier version, you might encounter compilation errors and might have to specify an earlier compiler version before starting the analysis.

For instance, if the Polyspace analysis configuration uses the version GCC 4.9 and some of the standard headers in your GCC version include the file `x86intrin.h`, you can see a compilation error such as this error:

```
/usr/lib/gcc/x86_64-linux-gnu/6/include/avx512bwintrin.h, line 2427:
                                    error: invalid type conversion
|    return (__m512i) __builtin_ia32_packssdw512_mask ((__v16si) __A,
|
```

You had to connect the error to the incorrect compiler version, and then explicitly set a different version. Now, the compiler version is automatically detected when you create a project from your build command.

Updated Bug Finder defect checkers

Summary: In R2019b, this defect checker has been updated.

Defect	Description	Update
`Pointer or reference to stack variable leaving scope`	Pointer to local variable leaves the variable scope	The checker now detects pointer escape via lambda expressions.

Compatibility Considerations

If you check for the defect mentioned above, you can see a difference in the number of issues found.

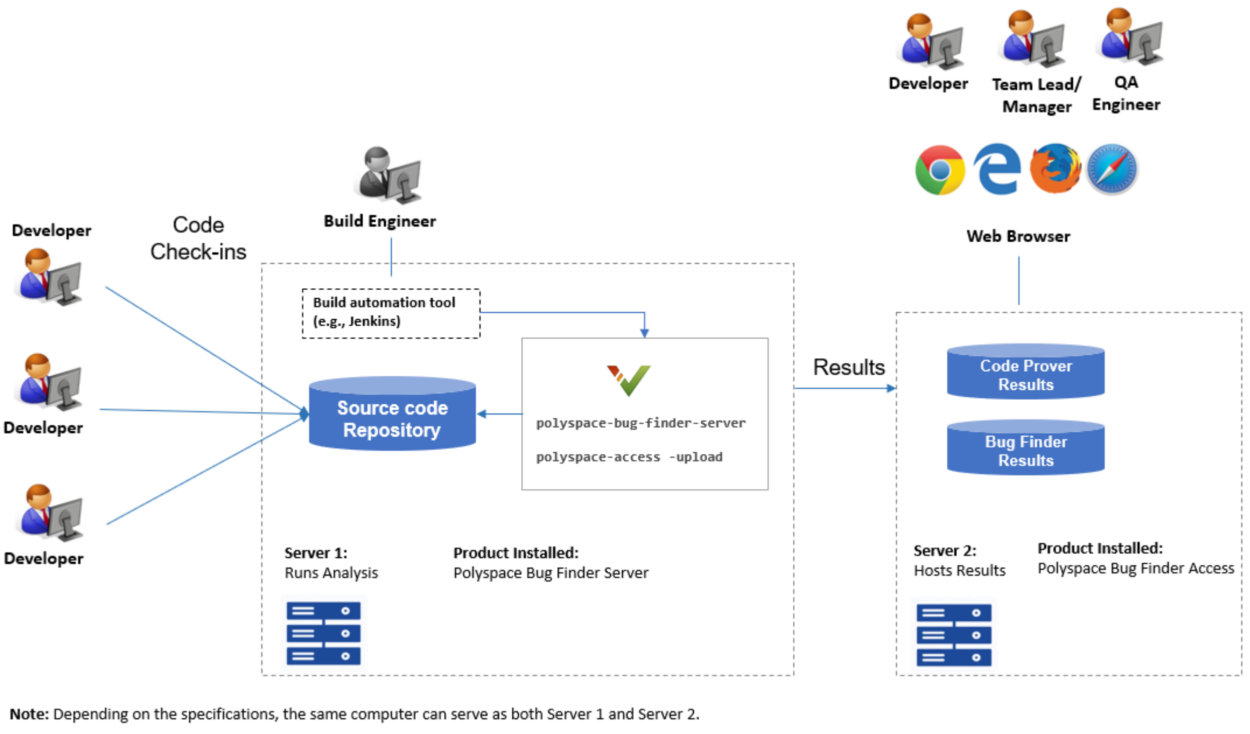
R2019a

Version: 3.0

New Features

Bug Finder Analysis Engine Separated from Viewer: Run Bug Finder analysis on server and view the results from multiple client machines

Summary: In R2019a, you can run Bug Finder on a server with the new product, Polyspace Bug Finder Server™. You can then host the analysis results on the same server or a second server with the product, Polyspace Bug Finder Access™. Developers whose code was analyzed (and other reviewers such as quality engineers and development managers) can fetch these results from the server to their desktops and view the results in a web browser, provided they have a Polyspace Bug Finder Access license.

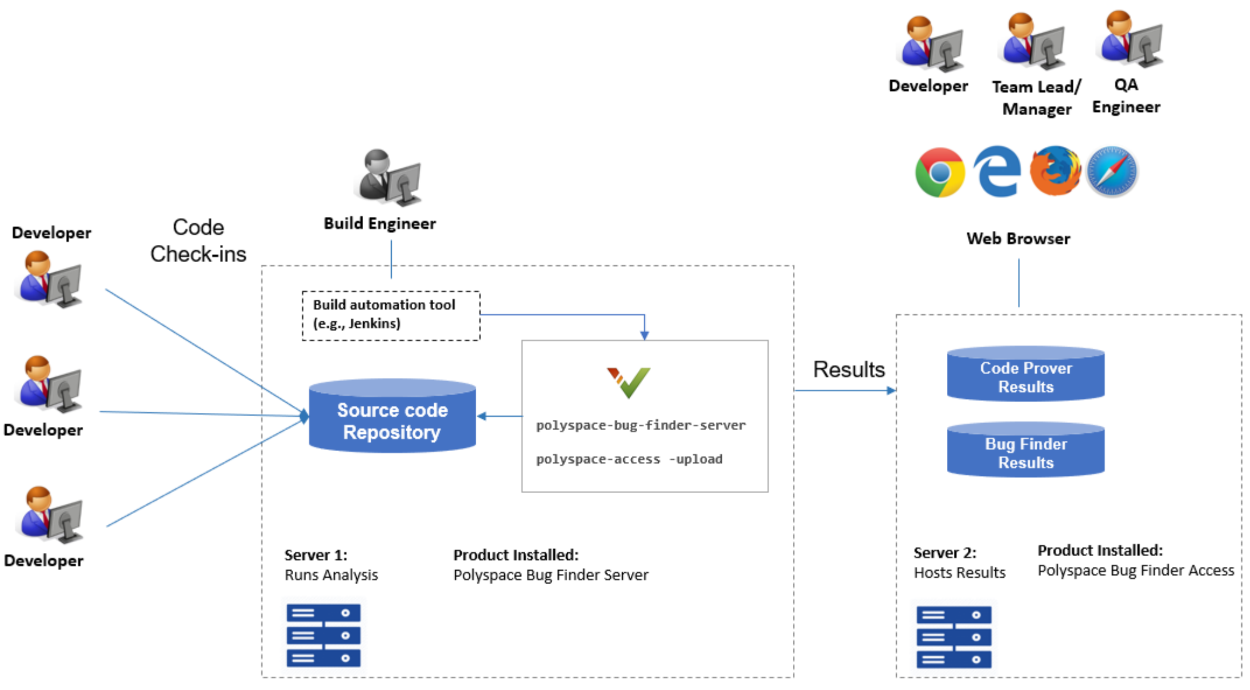

Developer
Team Lead/ Manager
QA Engineer
Web Browser
Build Engineer
Developer
Code Check-ins
Developer
Developer
Build automation tool (e.g., Jenkins)
Source code Repository
polyspace-bug-finder-server
polyspace-access -upload
Results
Code Prover Results
Bug Finder Results
Server 1: Runs Analysis
Product Installed: Polyspace Bug Finder Server
Server 2: Hosts Results
Product Installed: Polyspace Bug Finder Access
Note: Depending on the specifications, the same computer can serve as both Server 1 and Server 2.

Benefits: You can run the Bug Finder analysis on a few powerful server class machines but view the analysis results from many terminals.

With the desktop product, Polyspace Bug Finder, you have to run the analysis and view the results on the same machine. To view the results on a different machine, you need a second instance of a desktop product. The desktop products can now be used by individual developers on their desktops prior to code submission and the server products used after code submission. See Polyspace Products for Code Analysis and Verification.

Continuous Integration Support: Run Bug Finder on server class computers with continuous upload to Polyspace Access web interface

Summary: In R2019a, you can check for bugs, coding standard violations and other issues on server class machines as part of continuous integration. When developers submit code to a shared repository, a build automation tool such as Jenkins can perform the checks using the new Polyspace Bug Finder Server product. The analysis results can be uploaded to the Polyspace Access web interface for review. Each reviewer with a Polyspace Bug Finder Access license can login to the Polyspace Access web interface and review the results.



Note: Depending on the specifications, the same computer can serve as both Server 1 and Server 2.

See:

- Install Polyspace Server and Access Products
- Run Polyspace Bug Finder on Server and Upload Results to Web Interface

Benefits:

- *Automated post-submission checks*: In a continuous integration process, build scripts run automatically on new code submissions before integration with a code base. With the new product Polyspace Bug Finder Server, a Bug Finder analysis can be included in this build process. The analysis can run a specific set of Bug Finder checkers on the new code submissions and report the results. The results can be reviewed in the Polyspace Access web interface with a Polyspace Bug Finder Access license.
- *Collaborative review*: The analysis results can be uploaded to the Polyspace Access web interface for collaborative review. For instance:

  - Each quality assurance engineer with a Polyspace Bug Finder Access license can review the Bug Finder results on a project and assign issues to developers for fixing.
  - Each development team manager with a Polyspace Bug Finder Access license can see an overview of Bug Finder results for all projects managed by the team (and also drill down to details if necessary).

  For further details, see the release notes of Polyspace Bug Finder Access .

Continuous Integration Support: Set up testing criteria based on Bug Finder static analysis results

Summary: In R2019a, you can run Bug Finder on server class machines as part of unit and integration testing. You can define and set up testing criteria based on Bug Finder static analysis results.

For instance, you can set up the criteria that new code submissions must have zero high-impact defects before integration with a code base. Any submission with high-impact defects can cause a test failure and require code fixes.

See:

- `polyspace-bug-finder-server` for how to run Bug Finder on servers.
- `polyspace-access` for how to export Bug Finder results for comparison against predefined testing criteria.

If you use Jenkins for build automation, you can use the Polyspace plugin. The plugin provides helper functions to filter results based on predefined criteria. See Sample Scripts for Polyspace Analysis with Jenkins.

Benefits:

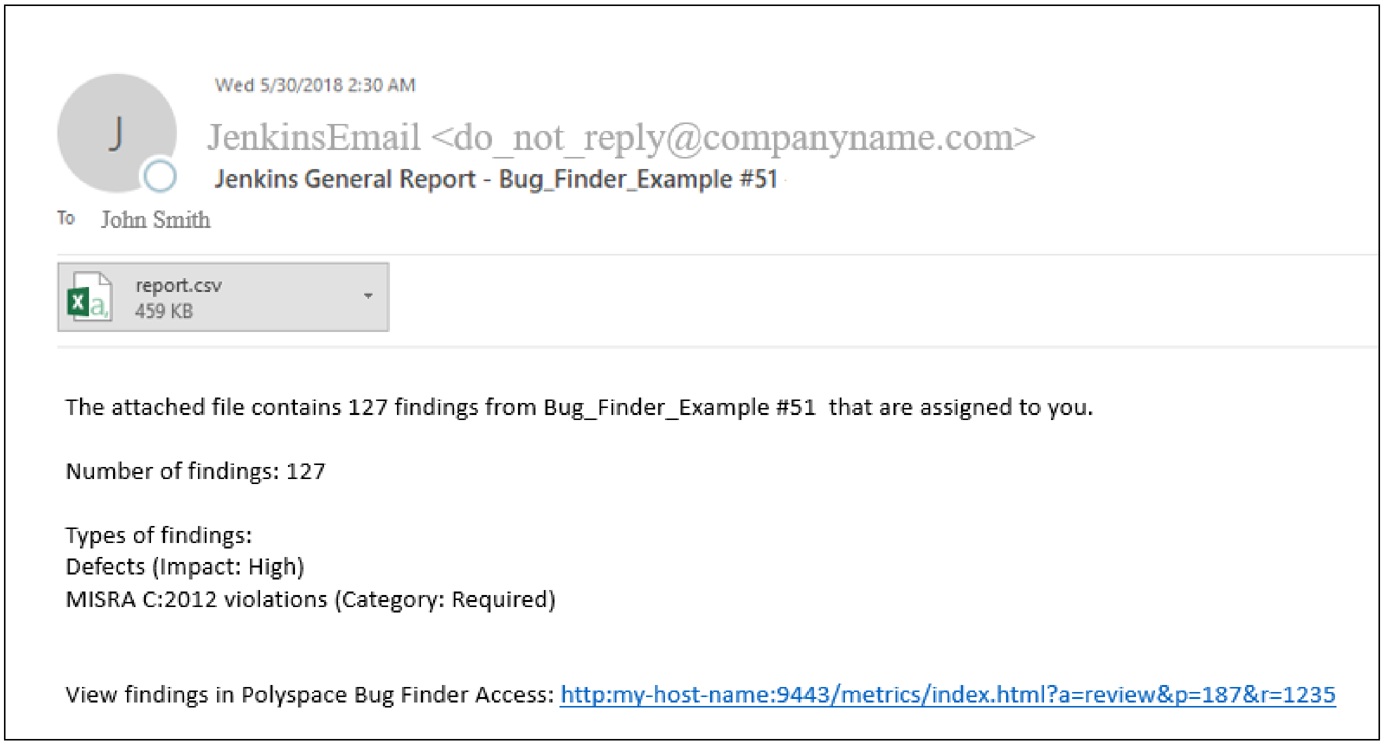
- *Automated testing*: After you define testing criteria based on Bug Finder results, you can run the tests along with regular dynamic tests. The tests can run on a periodic schedule or based on predefined triggers.
- *Prequalification with Polyspace desktop products*: Prior to code submission, to avoid test failures, developers can perform a pre-submit analysis on their code with the same criteria as the server-side analysis. Using an installation of the desktop product, Polyspace Bug Finder, developers can emulate the server-side analysis on their desktops and review the results in the user interface of the desktop product. For more information on the complete suite of Polyspace products available for deployment in a software development workflow, see Polyspace Products for Code Analysis and Verification.

  To save processing power on the desktop, the analysis can also be offloaded to a server and only the results reviewed on the desktop. See Install Products for Submitting Polyspace Analysis from Desktops to Remote Server.

Continuous Integration Support: Set up email notification with summary of Bug Finder results after analysis

Summary: In R2019a, you can set up email notification for new Bug Finder results. The email can contain:

- A summary of new results from the latest Bug Finder run only for specific files or modules.
- An attachment with a full list of the new results. Each result has an associated link to the Polyspace Access web interface for more detailed information.



Wed 5/30/2018 2:30 AM

JenkinsEmail <do_not_reply@companyname.com>

Jenkins General Report - Bug_Finder_Example #51

To John Smith

report.csv
459 KB

The attached file contains 127 findings from Bug_Finder_Example #51 that are assigned to you.

Number of findings: 127

Types of findings:
Defects (Impact: High)
MISRA C:2012 violations (Category: Required)

View findings in Polyspace Bug Finder Access: http:my-host-name:9443/metrics/index.html?a=review&p=187&r=1235

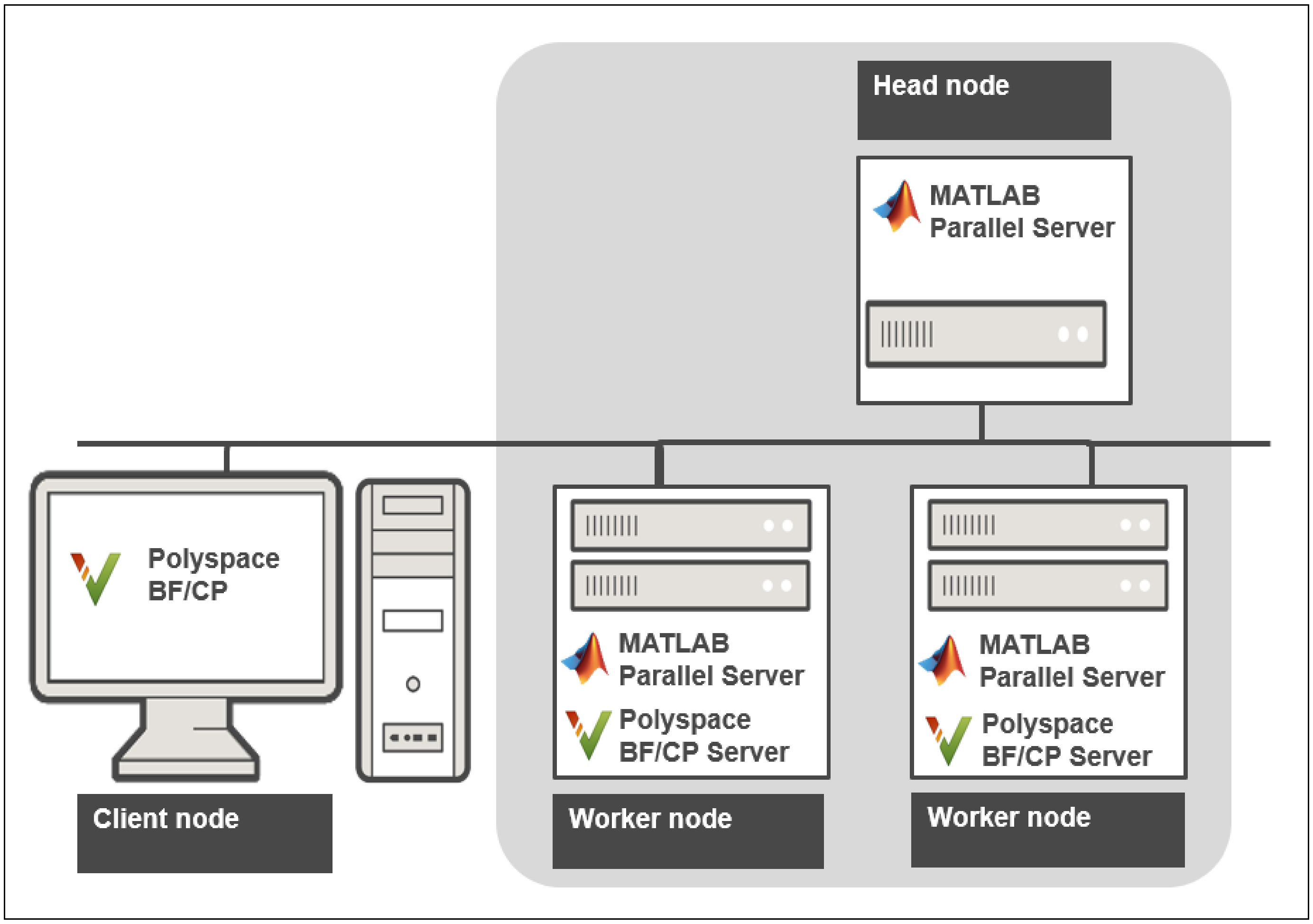
See Send E-mail Notifications with Polyspace Bug Finder Results.

Benefits:

- *Automated notification*: Developers can get notified in their e-mail inbox about results from the last Bug Finder run on their submissions.
- *Preview of Bug Finder results*: Developers can see a preview of the new Bug Finder results. Based on their criteria for reviewing results, this preview can help them decide whether they want to see further details of the results.
- *Easy navigation from e-mail summary to Polyspace Access web interface*: Each developer with a Polyspace Bug Finder Access license can use the links in the e-mail attachments to see further details of a result in the Polyspace Access web interface.

Offloading Polyspace Analysis to Servers: Use Polyspace desktop products on client side and server products on server side

Summary: In R2019a, you can offload a Polyspace analysis from your desktop to remote servers by installing the Polyspace desktop products on the client side and the Polyspace server products on the server side. After analysis, the results are downloaded to the client side for review. You must also install MATLAB Parallel Server on the server side to manage submissions from multiple client desktops.



See Install Products for Submitting Polyspace Analysis from Desktops to Remote Server.

Benefits: The Polyspace desktop products have a graphical user interface. You can configure options in the user interface with assistance from features such as auto-population of option arguments and contextual help. To save processing time on your desktop, you can then offload the analysis to remote servers.