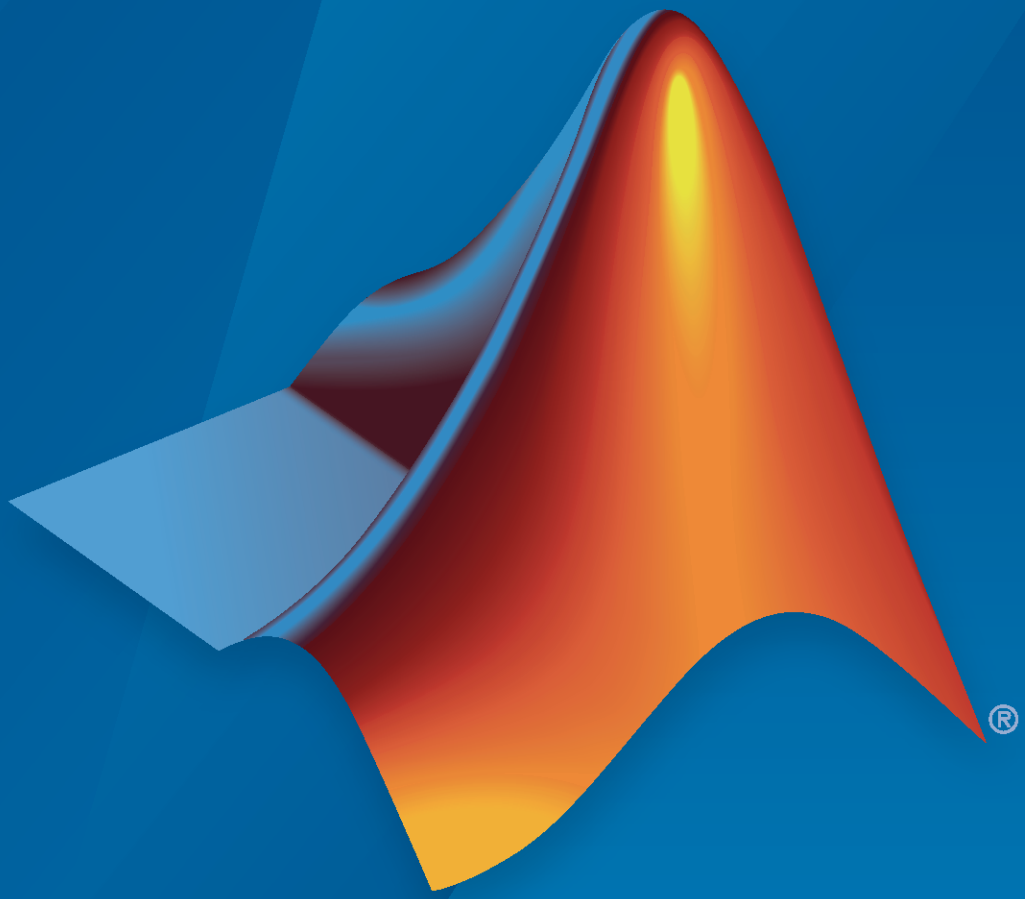
Polyspace® Bug Finder™ Server™ Release Notes



MATLAB®

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

R2020a

R2019b

R2019a

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, almost 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: Rec. 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”.

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

Summary: In R2020a, you can check for new issues and also see improved results for previous checkers.

New Checkers in R2020a

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

Updated Checkers in R2020a

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.

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.

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.
`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.

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.

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 and updated Bug Finder defect checkers

Summary: In R2019b, you can check for new issues and also see improved results for previous checkers.

New Checkers in R2019b

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

Updated Checkers in R2019b

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.

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.

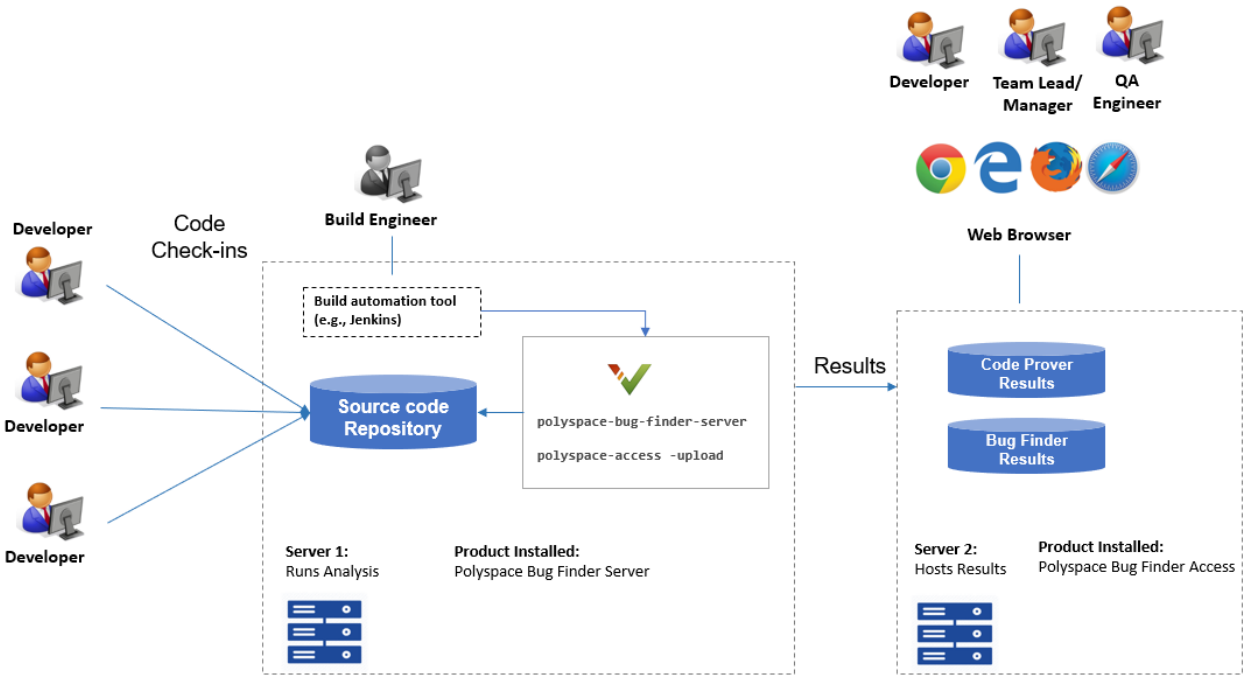
R2019a

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.



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

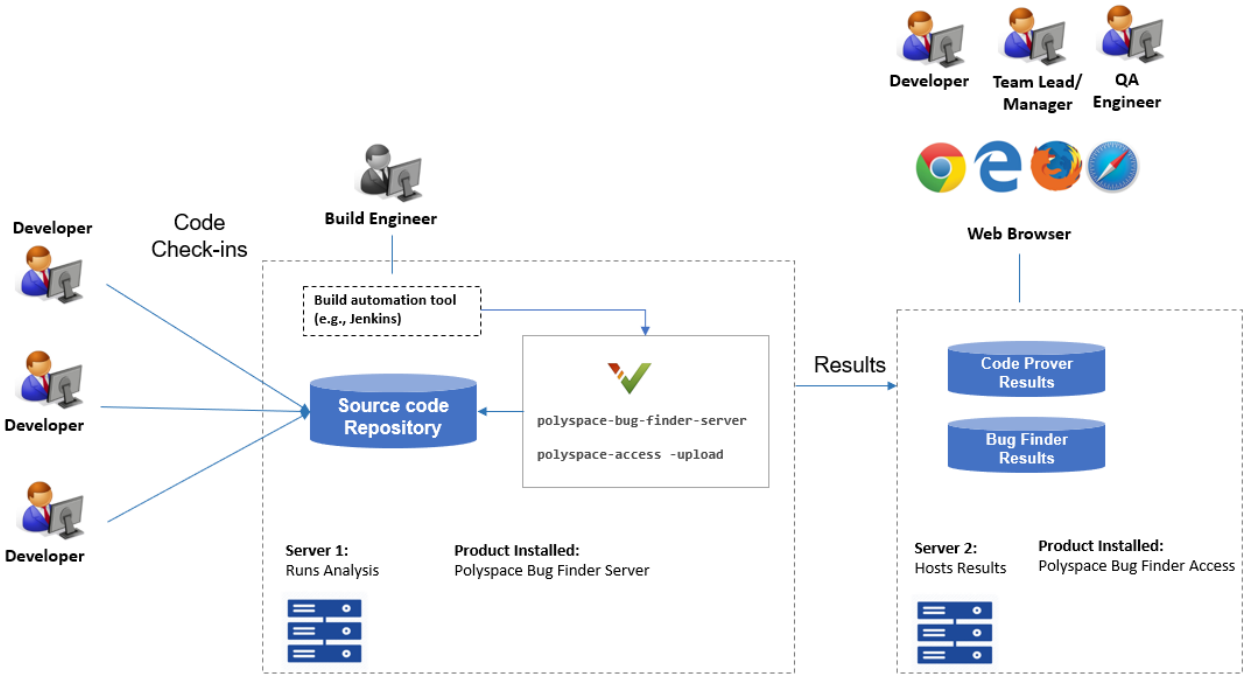
Benefits: You can run the Bug Finder analysis on a few powerful server class machines but view the analysis results from many terminals.

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:

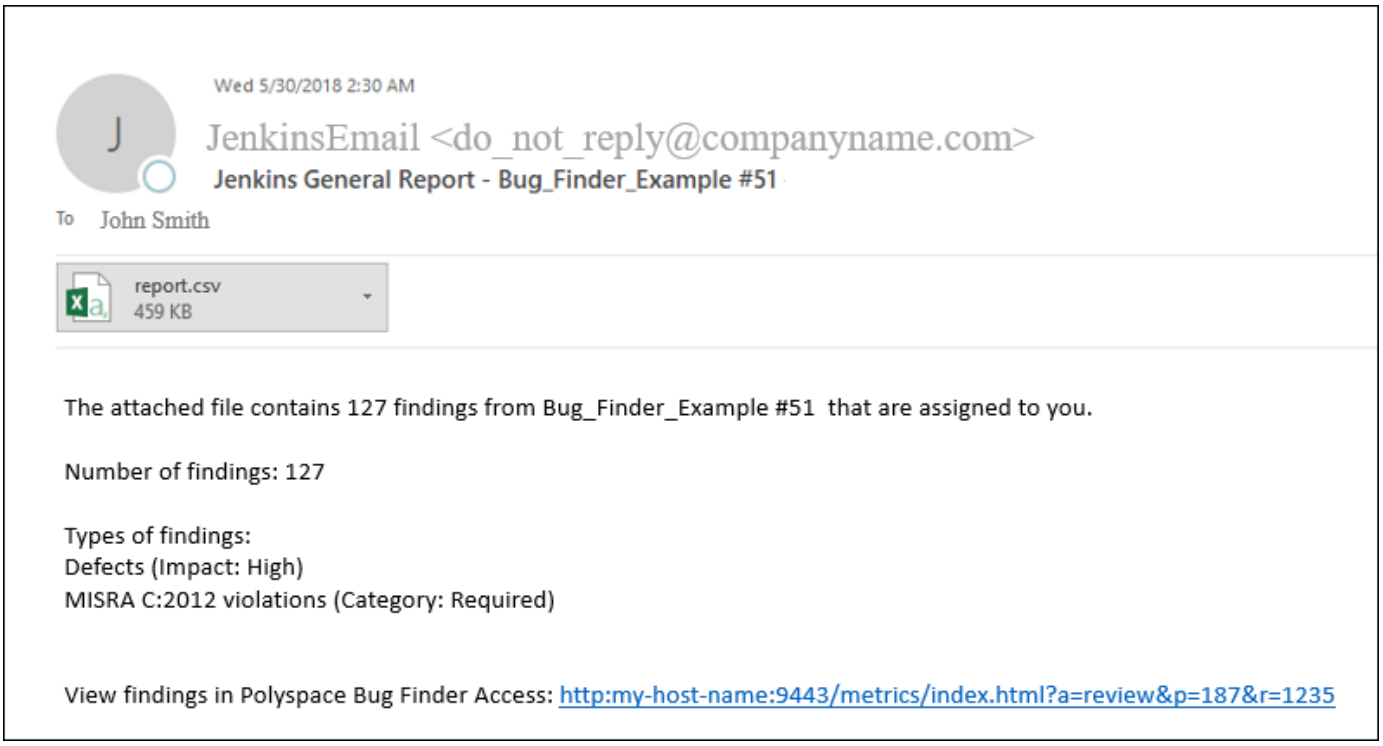
- *Automated testing*: After you define testing criteria based on Bug Finder results, you can run the tests along with regular dynamic tests. The tests can run on a periodic schedule or based on predefined triggers.
- *Prequalification with Polyspace desktop products*: Prior to code submission, to avoid 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.



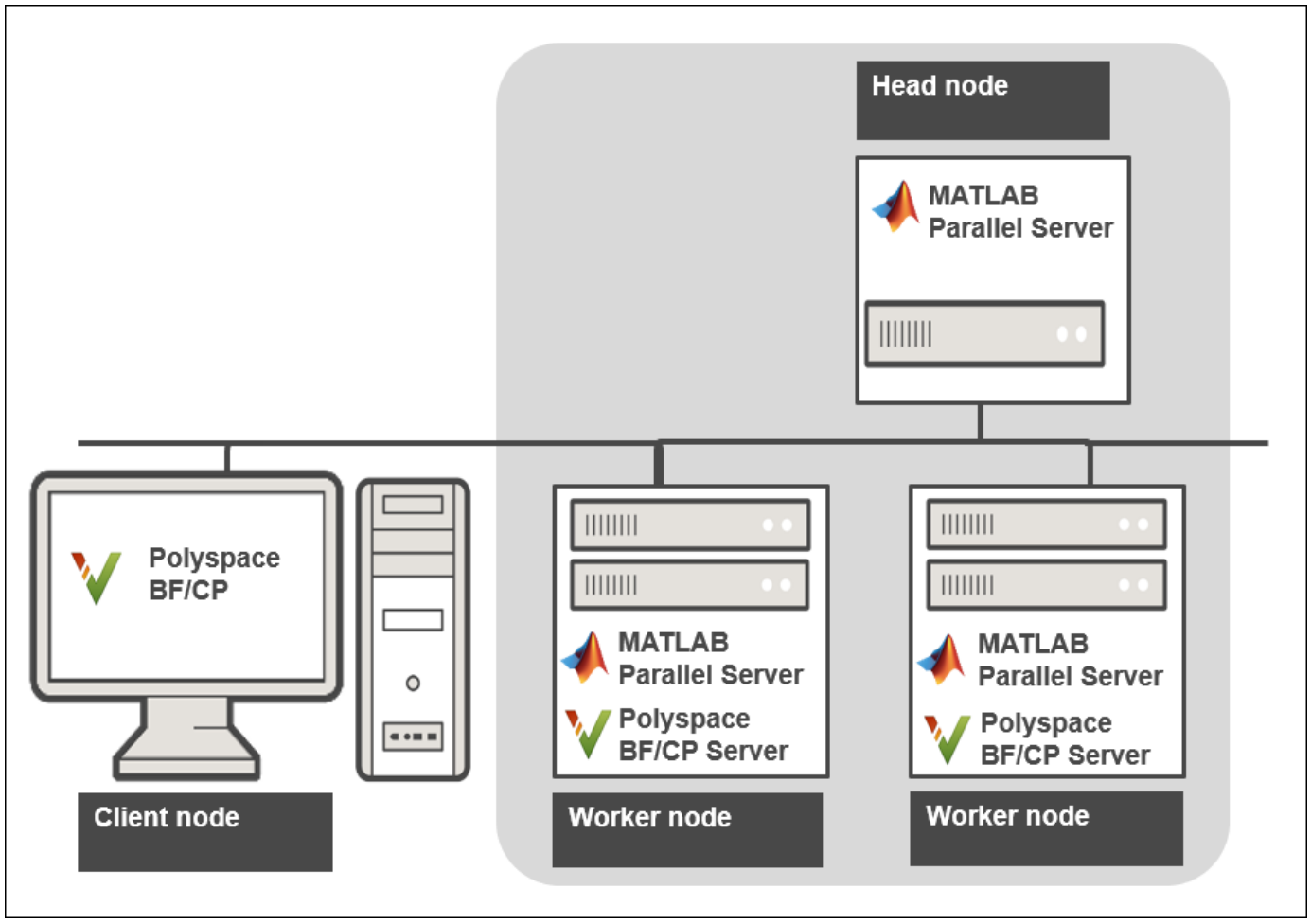
See Send E-mail Notifications with Polyspace Bug Finder Results.

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.