

Built-in Test Configurations

This topic describes the preconfigured "built-in" Test Configurations that are included with C++test.

C++test includes a set of preconfigured "built-in" Test Configurations representing most common test scenarios. You can further customize these configurations as needed by copying and modifying the built-in configurations, or by creating new user-defined configurations from scratch. User-defined Test Configurations can be placed in the User-defined or Team category. User-defined Test Configurations are stored on the local machine and are available for all tests performed by the local C++test installation. Team Test Configurations are stored on the team's Team Server and can be accessed by all team members.

Static Analysis Group

| Test Configuration | Description |
|---|--|
| Flow Analysis Standard | Detects complex runtime errors without requiring test cases or application execution. Defects detected include using uninitialized or invalid memory, null pointer dereferencing, array and buffer overflows, division by zero, memory and resource leaks, and dead code. This requires a special Flow Analysis license option. See Introducing Built-in Flow Analysis Test Configurations for more details on Flow Analysis Test Configurations. |
| Flow Analysis Fast | The fast configuration uses "Shallowest" depth of analysis and runs faster than the standard and aggressive configurations. The fast configuration finds a moderate amount of problems and prevents violation number explosion. See Introducing Built-in Flow Analysis Test Configurations for more details on Flow Analysis Test Configurations. |
| Flow Analysis Aggressive | The aggressive option reports any suspicious code as a violation. See Introducing Built-in Flow Analysis Test Configurations for more details on Flow Analysis Test Configurations. |
| CERT C Coding Standard | Checks rules for the CERT C Secure Coding Standard. This standard provides guidelines for secure coding. The goal is to facilitate the development of safe, reliable, and secure systems by, for example, eliminating undefined behaviors that can lead to undefined program behaviors and exploitable vulnerabilities. |
| CRules | Checks rules that enforce C best practices. |
| CWE-SANS Top 25 Most Dangerous Programming Errors | Checks for the 2011 CWE/SANS Top 25 Most Dangerous Software Errors— a list of the most widespread and critical errors that can lead to serious vulnerabilities in software. They are often easy to find, and easy to exploit. They are dangerous because they will frequently allow attackers to completely take over the software, steal data, or prevent the software from working at all. (http://cwe.mitre.org/top25/index.html) For more details, see 2011 CWE/SANS Top 25 Most Dangerous Software Errors Mapping . |
| DISA-STIG Coding Standard | Checks rules from DISA STIG. These rules are the United States Department of Defense's set of guidelines for secure development. |
| Effective C++ | Checks rules from Scott Meyers' "Effective C++" book. These rules check the efficiency of C++ programs. |
| Effective STL | Checks rules from Scott Meyers' "Effective STL" book. |
| Find Duplicated Code | Detects duplicated functions, code fragments, string literals, and #include directives. |
| HIS Source Code Metrics | Checks metrics required by the Herstellerinitiative Software (HIS) group. |
| ISO 26262 Recommended Rules | Checks rules recommended by the ISO 26262 standard. |
| Joint Strike Fighter | Checks rules that enforce the Joint Strike Fighter (JSF) program coding standards. |
| Metrics | Reports metrics statistics and detects metric values out of acceptable ranges. |
| MISRA C | Checks rules that enforce the MISRA C coding standards. |
| MISRA C 2004 | Checks rules that enforce the MISRA C 2004 coding standards. |
| MISRA C++ 2008 | Checks rules that enforce the MISRA C++ 2008 coding standards. |
| MISRA C 2012 | Checks rules that enforce the MISRA C 2012 coding standards. |
| OWASP Top 10 Security Vulnerabilities | Checks rules for the security issues referenced in the OWASP Top 10 Security Vulnerabilities (http://www.owasp.org/index.php/Top_10_2007) |

| | |
|--|--|
| Payment Card Industry Data Security Standard | Checks rules for the security issues referenced in section 6 of the Payment Card Industry Data Security Standard (PCI DSS) (https://www.pcisecuritystandards.org/security_standards/pci_dss.shtml) Issues detected include input validation (to prevent cross-site scripting, injection flaws, malicious file execution, etc.) and validation of proper error handling. |
| Parasoft's Recommended Rules | Checks the rules that are most likely to identify serious construction defects: coding constructs that cause application problems such as slow performance, memory leaks, security vulnerabilities, and so on. |
| Parasoft's Recommended FDA C++ Phase 1 | Checks the core set of rules recommended for complying with the FDA General Principles for Software Validation. |
| Parasoft's Recommended FDA C++ Phase 2 | Checks a broader set of rules recommended for complying with the FDA General Principles for Software Validation; recommended for organizations who have already implemented the phase 1 rule set. |
| Parasoft's Recommended FDA C++ Phase 3 | Checks a broader set of rules recommended for complying with the FDA General Principles for Software Validation; recommended for organizations who have already implemented the phase 2 rule set. |
| SAMATE Annex A Source Code Weaknesses | Checks code weaknesses listed in Annex A of NIST Software Assurance Metrics and Tool Evaluation (SAMATE) project (http://sama.te.nist.gov). |
| SecurityRules | Checks rules designed to prevent or identify security vulnerabilities. |
| Sutter-Alexandrescu | Checks rules based on the book "C++ Coding Standards," by Herb Sutter and Andrei Alexandrescu. |
| The Power of Ten | Checks rules based on Gerard J. Holzmann's article "The Power of Ten - Rules for Developing Safety Critical Code." (http://spinroot.com/gerard/pdf/Power_of_Ten.pdf) |

Unit Testing Group

| Test Configuration | Description |
|---|--|
| File Scope> Build Test Executable (File Scope) | Builds test executable for "trial builds." <i>Only the selected file(s) will be instrumented.</i> |
| File Scope> Collect Stub Information (File Scope) | Collects symbols data to populate the Stubs view. <i>Only the selected file(s) will be instrumented.</i> |
| File Scope> Debug Unit Tests (File Scope) | Executes unit tests under the debugger. <i>Only the selected file(s) will be instrumented.</i> |
| File Scope> Generate Stubs (File Scope) | Generates stubs for missing function and variable definitions. <i>Only the selected file(s) will be instrumented.</i> |
| File Scope> Run Unit Tests | Executes the available test cases. <i>Only the selected file(s) will be instrumented.</i> |
| Build Test Executable | Builds test executable for "trial builds." <i>All project files will be instrumented.</i> |
| Collect Stub Information | Collects symbols data to populate the Stubs view. <i>All project files will be instrumented.</i> |
| Debug Unit Tests | Executes unit tests under the debugger. <i>All project files will be instrumented.</i> |

| | |
|---------------------------------------|---|
| Generate Regression Base | Generates a baseline test suite that captures the project code's current functionality; to detect changes from this baseline, you run your evolving code base against this test suite on a regular basis. Outcomes are automatically verified. |
| Generate Stubs | Generates stubs for missing function and variable definitions. <i>All project files will be instrumented.</i> |
| Generate Test Suites | Generates test suites (without generating test cases) for the selected resources. |
| Generate Unit Tests | Generates unit tests for the selected resources. |
| Run Unit Tests | Executes the available test cases. <i>All project files will be instrumented.</i> |
| Run Unit Tests with Memory Monitoring | Executes the available test cases and collects information about memory problems. <i>All project files will be instrumented.</i> |

Application Monitoring Group

| Test Configuration | Description |
|--|---|
| Build Application with Coverage Monitoring | Builds the tested application with coverage monitoring enabled. |
| Build Application with Full Monitoring | Builds the tested application with coverage and memory monitoring enabled. |
| Build Application with Memory Monitoring | Builds the tested application with memory monitoring enabled. |
| Build and Run Application with Coverage Monitoring | Builds and executes the tested application with coverage monitoring enabled. |
| Build and Run Application with Full Monitoring | Builds and executes the tested application with coverage and memory monitoring enabled. |
| Build and Run Application with Memory Monitoring | Builds and executes the tested application with memory monitoring enabled. |

Embedded Systems Group

| Test Configuration | Description |
|--|--|
| Window Mobile> Build Test Executable for Windows Mobile | Builds a test executable that you need to manually transfer to the target device and run. This Test Configuration is very similar to the "Build Test Executable" Test Configuration; the only difference is that it is configured to use an external storage card to generate post-run artifacts (coverage and results). See Windows Mobile Support for details. |
| Window Mobile> Build and Run Test Executable for Pocket PC | Builds the test executable, then deploys it to the emulator and runs it. After execution completes, you need to close the emulator to prompt C++test to read and display test results. See Windows Mobile Support for details. |
| Window Mobile> Build and Run Test Executable for Smartphone | Builds the test executable, then deploys it to the emulator and runs it. After execution completes, you need to close the emulator to prompt C++test to read and display test results. See Windows Mobile Support for details. |
| Window Mobile> Build and Run Test Executable for Windows Mobile or Windows CE Using ActiveSync | Builds the test executable, then deploys it to the emulator and runs it. ActiveSync is used as a communication channel. To use this flow, both host and target machines must support ActiveSync. The target can be a real device connected in ways supported by ActiveSync, or it can be an emulator. See Windows Mobile Support for details. |

Utilities Group

| Test Configuration | Description |
|--------------------------|---|
| Load Test Results (File) | Used to collect test results via the file channel. By default, this configuration assumes that logs are located inside <code>\${cpptest: testware_loc}</code> . If needed, you can customize this location to any file system location that can be accessed from the C++test GUI. |

| | |
|---|---|
| Load Test Results (Sockets) | Used for "on the fly" collection of test results sent through TCP/IP sockets. It starts a java utility program to listen to and capture test results. You can customize the port numbers for test and coverage results. Port numbers are defined with the <code>results_port</code> and <code>coverage_port</code> properties. |
| Extract Library Symbols | Used to extract a list of symbols from external libraries (or object files). It should be used whenever C++test's standard algorithm for collecting information about symbols from binaries is not sufficient. For example if you use a Wind River DKM type of project, you may want to have all symbols from the VxWorks image collected in this way. You will probably need to enter the location of the binaries you want to extract symbols from, as well as the name of the nm-like utility that can be used to dump the content of library/object file. |
| Generate Stubs Using External Library Symbols | Used to generate stubs after the "Extract Library Symbols" Test Configuration has been run. It assumes that a file with a list of symbols from external libraries is stored in the project temporary data. |

Code Review Group

| Name | Scope | Code Review |
|--------------------------|--|---|
| Pre-Commit | Only files added or modified locally | For teams who want to review code <i>before</i> it is committed to source control. To use this Test Configuration, the Code Review Preference Show user assistant during scanner run setting must be enabled so that the author can designate the appropriate reviewer(s). See the Code Review for details. |
| Post-Commit (Template) | All project files modified in the previous day | For teams who want to review code <i>after</i> it is committed to source control. This Test Configuration must be duplicated and customized prior to use (e.g. to specify author-reviewer mappings). See Code Review for details. |
| Post-Commit (Assign All) | All project files modified in the previous day | For teams who want to review code <i>after</i> it is committed to source control. This Test Configuration can be used without customization. It includes a mapping for the local code review user; it assigns all revisions found in scope (for any author) to the current user. |

See [Configuring Test Configurations and Rules for Policies](#) to learn how to develop custom Test Configurations that are tailored to your projects and team priorities.

2011 CWE/SANS Top 25 Most Dangerous Software Errors Mapping

| CWE ID | CWE Name | Parasoft ID | Parasoft Name |
|---------|--|----------------------|---|
| CWE-89 | Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') | BD-SECURITY-TDSQL | Protect against SQL injection |
| CWE-78 | Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') | BD-SECURITY-TDCMD | Protect against command injection |
| CWE-120 | Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') | BD-PB-OVERFFM | Avoid buffer overflow due to defining incorrect format limits |
| | | BD-PB-OVERFNZT | Avoid overflow due to reading a not zero terminated string |
| | | BD-PB-OVERFWR | Avoid overflow when writing to a buffer |
| | | BD-SECURITY-OVERFWR | Avoid buffer write overflow from tainted data |
| CWE-22 | Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') | BD-SECURITY-TDFNAMES | Protect against file name injection |
| CWE-676 | Use of Potentially Dangerous Function | PB-37 | The unbounded functions of library shall not be used |
| | | SECURITY-11 | Avoid using unsecured shell functions that may be affected by shell metacharacters |
| | | SECURITY-12 | Avoid using unsafe string functions which may cause buffer overflows |
| | | SECURITY-13 | Avoid using unsafe string functions that do not check bounds |
| | | SECURITY-14 | Do not use <code>scanf</code> and <code>fscanf</code> functions without specifying variable size in format string |
| | | SECURITY-16 | Never use <code>gets()</code> |
| | | SECURITY-22 | Do not use <code>mbstowcs()</code> function |

| | | | |
|---------|--|----------------------|---|
| | | SECURITY-30 | Avoid using 'getpw' function in program code |
| | | SECURITY-31 | Do not use 'cuserid' function |
| CWE-327 | Use of a Broken or Risky Cryptographic Algorithm | SECURITY-02 | Avoid functions which use random numbers from standard C library |
| | | SECURITY-28 | Standard random number generators should not be used to generate randomness for security reasons |
| | | SECURITY-37 | Do not use weak encryption functions |
| CWE-131 | Incorrect Calculation of Buffer Size | BD-PB-ARRAY | Avoid accessing arrays out of bounds |
| | | BD-PB-OVERFRD | Avoid overflow when reading from a buffer |
| | | BD-SECURITY-ARRAY | Avoid tainted data in array indexes |
| | | MRM-45 | Do not use sizeof operator on pointer type to specify the size of the memory to be allocated via 'malloc', 'calloc' or 'realloc' function |
| CWE-134 | Uncontrolled Format String | SECURITY-05 | Avoid using functions printf/wprintf with only one variable parameter |
| | | SECURITY-08 | Avoid using functions fprintf/fwprintf with only two parameters, when second parameter is a variable |
| CWE-190 | Integer Overflow or Wraparound | BD-SECURITY-INTOVERF | Protect against integer overflow/underflow from tainted data |
| | | MISRA-051 | Evaluation of constant unsigned integer expressions should not lead to wrap-around |