SCALe Analysis of <NAME> Codebase

<NAME\_AUTHOR>

<DATE>

<NAME\_ORGANIZATION>

<URL\_ORGANIZATION>

Copyright 2017 Carnegie Mellon University. All Rights Reserved.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

This report was prepared for the SEI Administrative Agent AFLCMC/AZS 5 Eglin Street Hanscom AFB, MA 01731-2100

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

Internal use:\* Permission to reproduce this material and to prepare derivative works from this material for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.

External use:\* This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

\* These restrictions do not apply to U.S. government entities.

Carnegie Mellon® and CERT® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM17-0160

Table of Contents

[1 Introduction 1](#_Toc480994055)

[1.1 Code Overview 1](#_Toc480994056)

[2 Findings 1](#_Toc480994057)

[2.1 Future Work 1](#_Toc480994058)

[3 Analysis of Findings 1](#_Toc480994059)

[3.1 *Violation:* <ABBREVIATED RULE OR CODE FLAW NAME> 1](#_Toc480994060)

[3.2 *Violation:* <ABBREVIATED RULE OR CODE FLAW NAME> 1](#_Toc480994061)

[3.3 *Violation:* <ABBREVIATED RULE OR CODE FLAW NAME> 1](#_Toc480994062)

[3.4 *Violation:* <ABBREVIATED RULE OR CODE FLAW NAME> 1](#_Toc480994063)

[3.5 *Violation:* <ABBREVIATED RULE OR CODE FLAW NAME> 1](#_Toc480994064)

[4 Diagnostic Findings 1](#_Toc480994065)

[4.1 Confirmed Diagnostics 1](#_Toc480994066)

[4.1.1 Checkers 1](#_Toc480994067)

[5 Procedure 1](#_Toc480994068)

[5.1 SEI CERT Coding Rules 1](#_Toc480994069)

[5.1.1 Risk Assessment 1](#_Toc480994070)

[5.2 Diagnostic Categorization 1](#_Toc480994071)

[5.3 Static Analysis Tools 1](#_Toc480994072)

[5.3.1 MSVC /analyze 1](#_Toc480994073)

[5.3.2 PC-Lint 1](#_Toc480994074)

[5.3.3 Fortify 360 SCA 1](#_Toc480994075)

[5.3.4 Coverity Prevent 1](#_Toc480994076)

[5.3.5 Rosecheckers 1](#_Toc480994077)

[5.3.6 Cppcheck 1](#_Toc480994078)

[5.3.7 Other Tools 1](#_Toc480994079)

[5.4 History 1](#_Toc480994080)

[6 Procedure 1](#_Toc480994081)

[6.1 SEI CERT Coding Rules 1](#_Toc480994082)

[6.1.1 Risk Assessment 1](#_Toc480994083)

[6.2 Diagnostic Categorization 1](#_Toc480994084)

[6.3 Static Analysis Tools 1](#_Toc480994085)

[6.3.1 MSVC /analyze 1](#_Toc480994086)

[6.3.2 PC-Lint 1](#_Toc480994087)

[6.3.3 Fortify 360 SCA 1](#_Toc480994088)

[6.3.4 Coverity Prevent 1](#_Toc480994089)

[6.3.5 Rosecheckers 1](#_Toc480994090)

[6.3.6 Cppcheck 1](#_Toc480994091)

[6.3.7 Other Tools 1](#_Toc480994092)

[6.4 History 1](#_Toc480994093)

[7 Procedure 1](#_Toc480994094)

[7.1 SEI CERT Coding Rules 1](#_Toc480994095)

[7.1.1 Risk Assessment 1](#_Toc480994096)

[7.2 Diagnostic Categorization 1](#_Toc480994097)

[7.3 Static Analysis Tools 1](#_Toc480994098)

[7.3.1 FindBugs 1](#_Toc480994099)

[7.3.2 Fortify 360 SCA 1](#_Toc480994100)

[7.3.3 Coverity Prevent 1](#_Toc480994101)

[7.3.4 Other Tools 1](#_Toc480994102)

[7.4 History 1](#_Toc480994103)

[8 Procedure 1](#_Toc480994104)

[8.1 SEI CERT Coding Rules 1](#_Toc480994105)

[8.1.1 Risk Assessment 1](#_Toc480994106)

[8.2 Diagnostic Categorization 1](#_Toc480994107)

[8.3 Static Analysis Tools 1](#_Toc480994108)

[8.3.1 Perl::Critic 1](#_Toc480994109)

[8.3.2 B::Lint 1](#_Toc480994110)

[8.3.3 Other Tools 1](#_Toc480994111)

[8.4 History 1](#_Toc480994112)

[References 1](#_Toc480994113)

Figures

[Figure 1: Violations by Priority <MODIFY DATA IN THIS CHART TO REFLECT ACTUAL FINDINGS> 1](#_Toc480994040)

[Figure 2: Violations by Tool <MODIFY DATA IN THIS CHART TO REFLECT ACTUAL FINDINGS> 1](#_Toc480994041)

[Figure 3: Violations by CERT Rule <MODIFY DATA IN THIS CHART TO REFLECT ACTUAL FINDINGS> 1](#_Toc480994042)

[Figure 4: Rules and Recommendations for C 1](#_Toc480994043)

[Figure 5: CERT Secure Coding Priority and Levels 1](#_Toc480994044)

[Figure 6: Rules for C++ 1](#_Toc480994045)

[Figure 7: Rules and Recommendations for C 1](#_Toc480994046)

[Figure 8: CERT Secure Coding Priority and Levels 1](#_Toc480994047)

[Figure 9: Rules and Recommendations for Java 1](#_Toc480994048)

[Figure 10: CERT Secure Coding Priority and Levels 1](#_Toc480994049)

[Figure 11: Rules and Recommendations for Perl 1](#_Toc480994050)

[Figure 12: CERT Secure Coding Priority and Levels 1](#_Toc480994051)

Tables

[Table 1: Code Size Metrics 1](#_Toc480994034)

[Table 2: Code Size Metrics Headers 1](#_Toc480994035)

[Table 3: Audit Summary Statistics 1](#_Toc480994036)

[Table 4: Audit Summary Statistics Headers 1](#_Toc480994037)

[Table 5: Diagnostic Column Headers 1](#_Toc480994038)

[Table 6: Additional Diagnostic Column Headers 1](#_Toc480994039)

# Introduction

The <NAME> codebase is <COMPLETE HIGH-LEVEL DESCRIPTION OF CODEBASE HERE. This chapter introduces the codebase. It provides some simple statistics about the size of the codebase being audited. The purpose of this chapter is to outline precisely what code was audited and what code was ignored. Many codebases depend on libraries, which are provided in binary form. This enables the codebase to be built, but the libraries cannot be audited themselves unless their source code is also provided. In theory, any library can contain vulnerabilities that would compromise a system; consequently, every attached library should be audited. In practice, this is usually impractical. Consequently, this chapter indicates any un-audited libraries or ignored code>.

## Code Overview

Table 1 describes the size of this codebase and Table 2 explains the headers. This codebase consists of <NUMBER> modules, as listed in Table 1.

Table : Code Size Metrics

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Package | **Source** | **Files** | **kLoC** | **File Size** | **kSigLoC** |
| <NAME> | <NUMBER> | <NUMBER> | <NUMBER> | <NUMBER> | <NUMBER> |

Table : Code Size Metrics Headers

|  |  |
| --- | --- |
| **Heading** | **Definition** |
| Source | Size of source code files (in kilobytes) (inc blanklines & comments) |
| Files | Number of source files |
| kLoC | Lines of source code (divided by 1000) |
| Filesize | Lines of code per file |
| kSigLoC | Lines of significant source code without blank lines or comments |

# Findings

<This section summarizes the audit diagnostics that turned out to be true or probable. It consists mostly of graphs that detail which CERT rules were violated, which tools found the violations, and how severe the violations actually were. It also compares the codebase's metrics (how many violations, code size, etc.) against the average metrics for all codebases submitted to SCALe. This should give the codebase owners a rough idea of the quality of their code.>

Figure : Violations by Priority <MODIFY DATA IN THIS CHART TO REFLECT ACTUAL FINDINGS>

Key finding: <FILL IN TEXT SUMMARY OF KEY FINDING SHOWN IN FIGURE 1>

Figure : Violations by Tool <MODIFY DATA IN THIS CHART TO REFLECT ACTUAL FINDINGS>

Key finding: < FILL IN TEXT SUMMARY OF KEY FINDING SHOWN IN FIGURE 2>

Figure : Violations by CERT Rule <MODIFY DATA IN THIS CHART TO REFLECT ACTUAL FINDINGS>

As noted in Section 5, the priority field is the product of three metrics that measure the severity of the violation, the likelihood that the violation can be exploited, and the cost of remediating the violation. The maximum priority field is theoretically 27, indicating a severe vulnerability that is most likely to be exploited and is least expensive to fix.

The priority field is designed to indicate what we believe to be an optimal priority for fixing diagnostics. According to Figure 1, the maximum priority occurring in rule violation instance is <NUMBER>, and these violations are the most in need of mitigation, followed by the priority <NUMBER> diagnostics, and so forth.

We describe some of the most critical diagnostics in the next section. To comply with the *SEI CERT <LANGUAGE> Coding Standard*, these diagnostics, along with those in our previous report, must be brought to compliance with this standard, henceforth known as the CERT[[1]](#footnote-1)® rules, as described in Section 5.

Several other <LANGUAGE> codebases have been audited. Table 3 lists some relevant summary metrics about this codebase in comparison with the others, and Table 4 explains the summary statistics headers.

Table : Audit Summary Statistics

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Files** | **kLoC** | | **ksigLoC** | | **Rules** | | **True** | **Susp** | **FileDens** | | **LineDens** | |
| <NAME> | <NUMBER> | | <NUMBER> | | <NUMBER> | | <NUMBER> | <NUMBER> | <NUMBER> | | <NUMBER> | | <NUMBER> |
| *Average* | <NUMBER> | | <NUMBER> | | <NUMBER> | | <NUMBER> | <NUMBER> | <NUMBER> | | <NUMBER> | | <NUMBER> |
| *Std Dev* | <NUMBER> | | <NUMBER> | | <NUMBER> | | <NUMBER> | <NUMBER> | <NUMBER> | | <NUMBER> | | <NUMBER> |

Table : Audit Summary Statistics Headers

|  |  |
| --- | --- |
| **Heading** | **Definition** |
| Files | Number of source files |
| kLoC | Lines of source code (/÷1000) |
| ksigLoC | Lines of significant source code (/÷1000) (without blank lines and comments) |
| Rules | Number of CERT rules that were violated |
| True | Number of true violations |
| Susp | Number of suspicious violations |
| FileDens | Ratio of defects per file: diagnostics÷/files |
| LineDens | Ratio of defects per code size: diagnostics÷ksLOC |

Key finding: <FILL IN TEXT SUMMARY OF KEY FINDING SHOWN IN FIGURE 2>

## Future Work

The diagnostics in the spreadsheet are sorted from lowest level (L1) to highest level (L3), so the diagnostics that occur earlier in each spreadsheet are more urgent and easier to fix than the diagnostics that occur later. Therefore, we recommend attending to the diagnostics in the order in which they appear.

After the outstanding diagnostics are fixed, the code may be presented to the CERT Division for a second SCALe audit. The purpose of a second audit is to verify that all diagnostics were fixed and no new violations were introduced. A codebase with no remaining diagnostics qualifies for a certification that the code complies with the CERT rules.

The client, for several reasons, may choose not to modify code that has a valid diagnostic. Typically, when there are many diagnostics, some are marked as *suspicious*. Suspicious diagnostics have not been inspected by a human but are very similar to at least one true diagnostic that has been inspected by a human. The client may ignore such a suspicious diagnostic if they judge it to be a false positive.

Furthermore, some diagnostics indicate code that may or may not be vulnerable due to external circumstances. For example, many concurrency diagnostics would not apply to code that is never run in a multithreaded environment. Likewise, some diagnostics apply to code only when it is run on certain platforms (such as 64-bit Linux). These diagnostics may be ignored if the code is only to be run on platforms where the code is not vulnerable.

In each case, the organization running the code imposes a constraint on the code, and this constraint mitigates the violation. Such a constraint must be documented to explain why the code is permitted. When code is submitted for a second audit, such constraints must be submitted along with the code so that the auditors can appreciate why a diagnostic seems to have been ignored. If the auditor concurs, then the code can be certified as compliant and be subject to the constraints imposed by the diagnostic and documented by the client. For example, a codebase might be certified as CERT-compliant only when executed on 64-bit Linux.

# Analysis of Findings

This section provides an in-depth analysis of some of the confirmed diagnostics listed in the previous section. The following sections explain why the code in question violates the rule, but the sections do not attempt to explain the rules themselves because they are meant to be self-contained, and each rule provides ample rationale for its purpose. Each of the CERT rules has a page about it on the CERT wiki, and at the bottom of each page is a section where the public can post comments related to the rule. Issues about the validity of any rule should be posted to the rule’s Comments section. The CERT Division welcomes feedback about the rules and about the validity of each diagnostic.

|  |
| --- |
| *Violation:* <ABBREVIATED RULE OR CODE FLAW NAME> <FILEPATH INCLUDING FILENAME> has the following code snippet:  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE> ...  This code <This blurb should examine a single diagnostic and justify why the auditor considered it to be a violation. The purpose of this section is to convince the codebase owners that the report is credible; it showcases real vulnerabilities. Ideally, the diagnostics examined should be the most severe ones found. EXPLAIN The Consequence of violating the rule; that is, how the vulnerability could be exploited. Each blurb should indicate a violation of a different secure coding rule to showcase the breadth of the analysis.>  Consequently, this code violates CERT rule  <FULL CERT RULE NAME WITH HYPERLINK TO CERT WIKI PAGE FOR THAT RULE>  *Solution:* <ABBREVIATED SOLUTION NAME>  <TEXT HERE SHOULD EXPLAIN REMEDIATION; THAT IS, HOW TO FIX THE CODE.> |
| *Violation:* <ABBREVIATED RULE OR CODE FLAW NAME> <FILEPATH INCLUDING FILENAME> has the following code snippet:  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE> ...  This code <This blurb should examine a single diagnostic and justify why the auditor considered it to be a violation. The purpose of this section is to convince the codebase owners that the report is credible; it showcases real vulnerabilities. Ideally, the diagnostics examined should be the most severe ones found. EXPLAIN The Consequence of violating the rule; that is, how the vulnerability could be exploited. Each blurb should indicate a violation of a different secure coding rule to showcase the breadth of the analysis.>  Consequently, this code violates CERT rule  <FULL CERT RULE NAME WITH HYPERLINK TO CERT WIKI PAGE FOR THAT RULE>  *Solution:* <ABBREVIATED SOLUTION NAME>  <TEXT HERE SHOULD EXPLAIN REMEDIATION; THAT IS, HOW TO FIX THE CODE.> |
| *Violation:* <ABBREVIATED RULE OR CODE FLAW NAME> <FILEPATH INCLUDING FILENAME> has the following code snippet:  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE> ...  This code <This blurb should examine a single diagnostic and justify why the auditor considered it to be a violation. The purpose of this section is to convince the codebase owners that the report is credible; it showcases real vulnerabilities. Ideally, the diagnostics examined should be the most severe ones found. EXPLAIN The Consequence of violating the rule; that is, how the vulnerability could be exploited. Each blurb should indicate a violation of a different secure coding rule to showcase the breadth of the analysis.>  Consequently, this code violates CERT rule  <FULL CERT RULE NAME WITH HYPERLINK TO CERT WIKI PAGE FOR THAT RULE>  *Solution:* <ABBREVIATED SOLUTION NAME>  <TEXT HERE SHOULD EXPLAIN REMEDIATION; THAT IS, HOW TO FIX THE CODE.> |
| *Violation:* <ABBREVIATED RULE OR CODE FLAW NAME> <FILEPATH INCLUDING FILENAME> has the following code snippet:  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE> ...  This code <This blurb should examine a single diagnostic and justify why the auditor considered it to be a violation. The purpose of this section is to convince the codebase owners that the report is credible; it showcases real vulnerabilities. Ideally, the diagnostics examined should be the most severe ones found. EXPLAIN The Consequence of violating the rule; that is, how the vulnerability could be exploited. Each blurb should indicate a violation of a different secure coding rule to showcase the breadth of the analysis.>  Consequently, this code violates CERT rule  <FULL CERT RULE NAME WITH HYPERLINK TO CERT WIKI PAGE FOR THAT RULE>  *Solution:* <ABBREVIATED SOLUTION NAME>  <TEXT HERE SHOULD EXPLAIN REMEDIATION; THAT IS, HOW TO FIX THE CODE.> |
| *Violation:* <ABBREVIATED RULE OR CODE FLAW NAME> <FILEPATH INCLUDING FILENAME> has the following code snippet:  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE>  <LINE NUMBER> <CODE SNIPPET LINE> ...  This code <This blurb should examine a single diagnostic and justify why the auditor considered it to be a violation. The purpose of this section is to convince the codebase owners that the report is credible; it showcases real vulnerabilities. Ideally, the diagnostics examined should be the most severe ones found. EXPLAIN The Consequence of violating the rule; that is, how the vulnerability could be exploited. Each blurb should indicate a violation of a different secure coding rule to showcase the breadth of the analysis.>  Consequently, this code violates CERT rule  <FULL CERT RULE NAME WITH HYPERLINK TO CERT WIKI PAGE FOR THAT RULE>  *Solution:* <ABBREVIATED SOLUTION NAME>  <TEXT HERE SHOULD EXPLAIN REMEDIATION; THAT IS, HOW TO FIX THE CODE.> |

# Diagnostic Findings

The analysis results are provided by a spreadsheet with sevaral worksheets. Because of its size, the spreadsheet is provided separately from this document. This section describes its contents.

## Raw Data

The leftmost worksheet, named “raw”, contains all the diagnostics that were provided by the various static analysis tools. This includes flagged nonconformities that were personally verified by analysts to be violations of the CERT rules (“true” positives). It also includes flagged nonconformities that have not been inspected by a human; however, each such diagnostic was produced by a checker that also produced a true violation (“suspicious” diagnostics). It also includes false positives. The spreadsheet is a table, and thus can be sorted by any field, or filtered on any value. For example, to filter out all but the true positives, select the filter next to the **verdict** entry (cell 1B), unselect *Select All*, and select *TRUE*.

The columns of the raw worksheet are listed in Table 5:

Table 5: Diagnostic Column Headers

|  |  |
| --- | --- |
| **Header** | **Definition** |
| ID | Unique ID for each diagnostic |
| Verdict | Indicates if diagnostic is true or false |
| Path | Path name to the directory containing the source file |
| Line | Line number where violation occurs |
| Message | Diagnostic message describing the violation |
| Checker | Short string indicating the category of the error (Each tool uses its own error IDs, but some do not provide any.) |
| Tool | Tool that identified the diagnostic |
| Rule | ID of the CERT guideline that is violated |
| Title | Name of the CERT guideline |
| Severity | Potential consequences of violating the CERT rule |
| Likelihood | Likelihood that violation of the rule results in an exploitable vulnerability |
| Remediation Cost | Estimate of the difficulty of mitigating the diagnostic |
| Priority | Overall priority of the diagnostic’s rule |
| Level | Rule’s priority level |

The “raw” worksheet is currently sorted from lowest level (L1) to highest level (L3); hence the diagnostics that occur higher in the table are more urgent than the diagnostics that occur lower. Consequently, we recommend attending to the diagnostics in the order in which they appear, first mitigating the true diagnostics and then the suspicious diagnostics.

### Checkers

Each static analysis tool provides a set of checkers. A checker is considered to be a routine that issues one type of diagnostic. Multiple checkers may test for the same problem but in different ways. Some tools provide error IDs, indicating the category of error they diagnose. When a tool provides error IDs, we assume each distinct error ID represents a distinct checker.

Other tools provide no error IDs, however, and in these cases we search for patterns in the tools’ message strings. Our usual approach is to apply a regular expression match to the message and associate each unique regular expression with a checker. For instance, the GCC compiler uses no error IDs, but many of its error messages are unchanging strings, such as

Example.c:111: warning: comparison between signed and unsigned integer expressions

Other error messages may include a variable name, such as

Example.c:111: warning: 'int foo()' declared 'static' but never defined

Such strings can be easily identified and captured using regular expressions.

All diagnostics, except for those identified manually, will have an associated checker.

## Other Worksheets

The other worksheets in the spreadsheet provide various tabular summaries of the data on the “raw” worksheet, along with several useful charts. The tables and charts can be updated by right-clicking their upper left corner and selecting *Refresh*. Consequently, the spreadsheet can be used to measure progress; as code is modified, fixed diagnostics can be removed, or edited, with the worksheets reflecting the remaining diagnostics.

# Procedure

C programs can be analyzed by an extensive number of static analysis (SA) tools. Our experience with static analysis tools has led us to the conclusion that each SA tool has its own strengths and weaknesses, and every tool can detect faults undetectable by other tools. The National Security Agency Center for Assured Software (NSA CAS) ran similar experiments with C/C++ and Java static analysis tools (Willis and Britton 2011), and they came to the same conclusion. Consequently, running only one SA tool is likely to miss many faults that other tools can detect.

We therefore employ a coverage analysis technique, where we use several SA tools to detect vulnerabilities and merge their results. This technique has several advantages; the biggest one being that we minimize the risk of overlooking critical vulnerabilities (that is, false negatives). Because of the different strengths of different tools, we can also gain new perspectives on vulnerabilities identified by multiple analyzers.

Many tools rely on the assumption that it is more prudent for an SA tool, when encountering some questionable code, to report it as a potential vulnerability than to ignore it. This assumption also enables a security analyst to manually inspect the code and confirm the vulnerability or eliminate it. It minimizes the possibility of ‘false negatives’, that is, uncaught vulnerabilities. However, it does increase the number of false positives; that is, code constructs that might be vulnerable, but turn out to be perfectly legitimate when taken in their total context.

Several tools yield many false positives. Validating each of these diagnostics requires an inspection of the code in question, but sometimes it is necessary to inspect the entire method or class containing the code, or all methods that invoke the method containing the questionable code. Consequently, an auditor has no hope of thoroughly inspecting each and every diagnostic that may be generated by an automated SA tool.

## SEI CERT Coding Rules

An essential element of secure coding in any programming language is well-documented and enforceable coding standards. Coding standards encourage programmers to follow a uniform set of rules and guidelines determined by the requirements of the project and organization, rather than by the programmer's familiarity or preference. Once established, these standards can be used as a metric to evaluate source code (using manual or automated processes).

The CERT Division has published *The SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition)*. This book provides rules and recommendations for secure coding in the C programming language. The goal of these rules and recommendations is to eliminate insecure coding practices. The application of the secure coding standard will lead to higher quality systems that are robust and more resistant to attack. This coding standard affects the wide range of products coded in C, such as PCs, game players, mobile phones, home appliances, and automotive electronics. It is designed specifically for code conforming to C11, with some support for POSIX and Windows. The CERT Division provides certification for code that is conformant with *The SEI CERT C Coding Standard*. The standard is available at the following web address:

<https://www.securecoding.cert.org/confluence/x/HQE>

This standard consists of nearly 100 rules, and is accompanied by many recommendations. Coding practices are defined to be rules when the following conditions are met:

1. Violation of the coding practice is likely to result in a [security flaw](https://www.securecoding.cert.org/confluence/display/seccode/BB.+Definitions#BB.Definitions-securityflaw) that may result in an exploitable [vulnerability](https://www.securecoding.cert.org/confluence/display/seccode/BB.+Definitions#BB.Definitions-vulnerability).
2. Conformance to the coding practice can be determined through automated analysis, formal methods, or manual inspection techniques.

Implementation of the secure coding rules defined in this standard are necessary (but not sufficient) to ensure the security of software systems developed in the C programming language.

Recommendations are guidelines or suggestions. Coding practices are defined to be recommendations when all of the following conditions are met:

1. Application of the coding practice is likely to improve system security.
2. One or more of the requirements necessary for a coding practice to be considered a rule cannot be met.

The set of recommendations that a particular development effort adopts depends on the security requirements of the final software product. Projects with high-security requirements can dedicate more resources to security and are consequently likely to adopt a larger set of recommendations.

To ensure that the source code conforms to this secure coding standard, it is necessary to have measures in place that check for rule violations. The most effective means of achieving this conformance is to use one or more static analysis tools. Where a rule cannot be checked by a tool, then a manual review is required.

Figure 4 illustrates a breakdown of the current rules and recommendations provided by the standard.

Figure : Rules and Recommendations for C

### Risk Assessment

Each guideline has an assigned priority. Priorities are assigned using a metric based on Failure Mode, Effects, and Criticality Analysis (FMECA) [[IEC 60812](https://www.securecoding.cert.org/confluence/display/seccode/AA.+Bibliography#AA.Bibliography-IEC608122006)]. Three values are assigned for each guideline on a scale of 1 to 3 for

**severity** - how serious are the consequences of the guideline being ignored

1 = low (denial-of-service attack, abnormal termination)

2 = medium (data integrity violation, unintentional information disclosure)

3 = high (run arbitrary code, privilege escalation)

**likelihood** - how likely is it that a [flaw](https://www.securecoding.cert.org/confluence/display/java/BB.+Definitions#BB.Definitions-securityflaw) introduced by ignoring the guideline could lead to an exploitable vulnerability

1 = unlikely

2 = probable

3 = likely

**remediation cost** - how expensive it is to comply with the guideline

1 = high (manual detection and correction)

2 = medium (automatic detection and manual correction)

3 = low (automatic detection and correction)

The three values are then multiplied together for each guideline. This product provides a measure that can be used in prioritizing the application of the guidelines. These products range from 1 to 27. Guidelines with a priority in the range of 1-4 are level 3 guidelines, 6-9 are level 2, and 12-27 are level 1. As a result, it is possible to claim level 1, level 2, or complete compliance (level 3) with a standard by implementing all guidelines in a level, as shown in Figure 5.

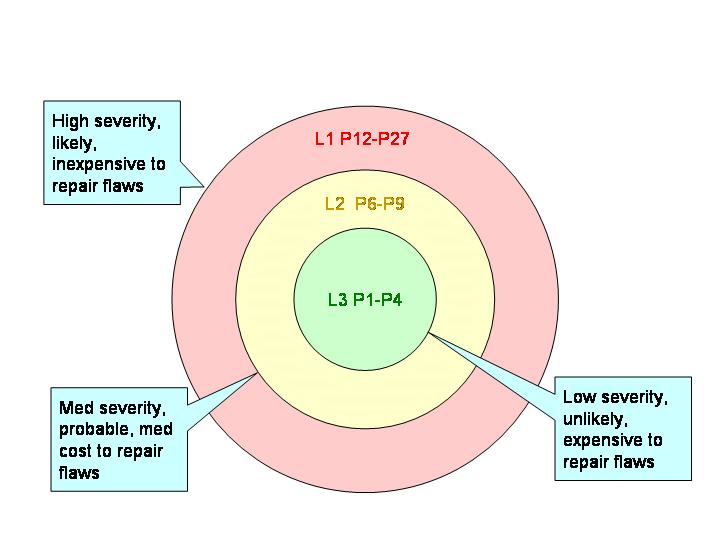


Figure : CERT Secure Coding Priority and Levels

## Diagnostic Categorization

Fortunately, many vulnerabilities rely on a relatively small handful of errors in coding technique, and many SA tools rely on a handful of heuristics to identify vulnerabilities. SA tools typically provide their own categorization of diagnostics and often assign a unique identifier for each diagnostic category. Furthermore, the diagnostics produced by SA tools can be easily associated with CERT rules, where a valid diagnostic indicates a violation of the associated CERT rule. While our SA tools produced many diagnostics, these diagnostics could be classified into violations of a few CERT rules.

Therefore, our approach involves collecting all diagnostics produced by all of the SA tools at our disposal and classifying them by the secure coding guideline they can be associated with. For each secure coding guideline, we then examine a handful of diagnostics. Any diagnostic that turns out to be a false positive is removed immediately. Any diagnostic that turns out to be a *true positive*, that is, indicates a true vulnerability in the code, is added to a table of confirmed vulnerabilities. We examine diagnostics for each guideline until we exhaust all of the diagnostics for that guideline, or until we have found a true positive. Using this process, we can produce a very small set of representative confirmed diagnostics, plus a large set of unconfirmed diagnostics. For every unconfirmed diagnostic, there exists at least one confirmed diagnostic with the same properties.

Some diagnostics are labeled ‘suspicious’. This label indicates that a diagnostic might conceivably be True under certain circumstances, but it was not inspected by an auditor. These uninspected portions of code may or may not be vulnerable to exploits. The code might actually be safe but difficult to analyze. It might actually be safe to use in certain limited contexts, and unsafe in others.

In any case, the code merits attention, and should probably be modified (depending on an organization’s priorities, funding, and effort availability). Without a fix, it is likely that the code may be passed to a maintainer who fails to understand the code and makes incorrect assumptions about its security. Incorrect assumptions increase the maintenance costs of the code, as the maintainer might modify it unnecessarily or might use it improperly, creating one or more new vulnerabilities.

This report provides the complete table of confirmed diagnostics, providing details associated with each. It also provides a similar table of suspicious diagnostics.

## Static Analysis Tools

We have employed the following SA tools, as described below:

### MSVC /analyze

Several editions of Microsoft Visual C++ provide a built-in static analysis tool. This tool includes MSVC 2008 Team Edition, and several editions of MSVC 2010. It is named *analyze mode* because of the /analyze option that is fed to the Microsoft C++ compiler commad. This tool can be enabled by turning on a switch called *Enable Code Analysis*. Consequently, any C/C++ program compiled by Visual Studio will be examined by the SA tool.

### PC-Lint

PC-Lint is a commercial static-analysis tool produced by Gimpel Software for the C and C++ programming languages. First released in 1985, it is supported on all versions of Windows, as well as MS-DOS and OS/2. It provides a command-line interface, but can also be integrated into many IDEs as an external tool, including Microsoft Visual Studio. It provides references to several coding guidelines such as MISRA-C (both 2004 and 2008 editions). More information on PC-Lint is available at the following link:

<http://www.gimpel.com>

### Fortify 360 SCA

Fortify 360 is a commercial product developed by Fortify Software. The product provides an extensive suite of tools for software security assurance. We focused on the source code analysis (SCA) tool. It can be used to analyze software written in C; C++; Java; .NET; ASP.NET; ColdFusion; "Classic" ASP, PHP, VB6, VBScript, JavaScript, PL/SQL, T-SQL and COBOL; as well as configuration files. More information on Fortify Software is available at the following link:

<http://www.fortify.com>

### Coverity Prevent

Coverity Prevent TM is a commercial product developed by Coverity, Inc. The product also provides an extensive suite of tools for software security assurance. We focused on the Coverity Static Analysis tool, which can be used to analyze software written in C, C++, Java, or C#. We also utilized the Coverity Integrity Manager, a web-based framework for viewing the results of Coverity Static Analysis. It provides a rich detail of each diagnostic found, including multiple locations in the source code that serve to create the diagnostic. More information on Coverity is available at the following link:

<http://www.coverity.com>

### Rosecheckers

The Rosecheckers project has been internally developed at the CERT Division to provide a static analysis tool for analyzing C and C++ code. The project was designed to enforce the rules in the *CERT C Secure Coding Standard*. Each rule in the standard that can be statically analyzed has one or more code checkers as part of the Rosecheckers project. The source for the Rosecheckers project is freely downloadable at the following link:

[http://rosecheckers.sourceforge.net](http://rosecheckers.sourceforge.netwww.coverity.com)

The website also provides a virtual machine containing a complete build of the Rosecheckers project on Linux.

The Rosecheckers project leverages the Compass/ROSE project developed at Lawrence Livermore National Labs. This project provides a high-level API for accessing the abstract syntax tree (AST) of a C or C++ source code file. More information on Compass/ROSE is available at the following link:

<http://rosecompiler.org>

### Cppcheck

Cppcheck is a free static analysis tool for C and C++.  More information about cppcheck is available at the following link:

<http://cppcheck.sourceforge.net/>

### Other Tools

Most compilers provide warnings for questionable code. Consequently, a compiler can serve as a simple SA tool, although compilers provide significantly fewer diagnostics than dedicated tools. Furthermore, several SA tools require the software to be compiled in order to function. Coverity, for instance, operates by monitoring a build as it progresses, and running its analysis on each file as it is compiled. Consequently, a program that cannot be completely built cannot be completely analyzed by Coverity.

Because of this, compilation of the software is a crucial first step, and we harvest any diagnostics produced by the compiler and perform the same analysis on them as we do for other SA tools.

Finally, auditors have the ability to provide diagnostics manually, if no tool reports them. It should be emphasized that manual inspection was not a primary procedure in this analysis; it was performed for two purposes: (1) to gain an intuitive overview of the code, and (2) to validate diagnostics produced by the SA tools. Nonetheless, a few diagnostics were might be noted during manual inspection; hence they are included in this report.

## History

A [SCALe](http://www.cert.org/secure-coding/products-services/scale.cfm) audit is a component of quality assurance for a codebase. It is often useful for a codebase to undergo an iterative process of SCALe audits and diagnostic mitigations; this is usually necessary for the codebase to comply with a CERT secure coding standard. Consequently, a codebase might be analyzed with SCALe multiple times. Sometimes the diagnostics reported in a previous audit may remain unfixed for various technical or business reasons. Furthermore, the SCALe process does not report to clients any diagnostics that are known to be false positives, and consequently code that produces such diagnostics is not modified, causing the false diagnostics to recur in subsequent audits.

For any codebase, a SCALe audit provides results that are significant and useful for any future audit of the codebase. When conducting an audit of any codebase that has undergone a previous audit, we can use the list of diagnostics from the previous audit, including any diagnostics that were discovered to be false positives and not presented to the client. If a diagnostic appears in a previous audit and was studied, then the information learned during the previous audit serves as a hint as to the diagnostic’s validity in the current audit. For example, a diagnostic that was revealed to indicate a true vulnerability in a previous audit is likely to still be true in the current audit. It would not be prudent to judge it true automatically. But if it is in a list of a hundred diagnostics, and was revealed to be true in a previous audit, we could choose to examine it first in the current audit. If it is still true, we could mark it so and proceed to the next batch of diagnostics.

SCALe uses a procedure to cascade diagnostic information from a previous analysis into a current analysis. The procedure requires the previous SCALe analysis results, the automatically-generated diagnostics for the new codebase, and the source code for both the old and current versions of the codebase. The procedure is as follows:

1. *Compute the differences between the old and new codebases.* This is easily accomplished using the UNIX diff(1) command. It might require some renaming of files. For example, if the old codebase has a directory named src-1.1, but the new codebase has the same directory named src-1.2, then the filename differences should be resolved. We are less interested in files that have been added, deleted, or moved around in the codebase, and we are more interested in source code files whose lines have been modified.
2. *Gather the old diagnostics, including false positives.* This may involve identifying diagnostics that were unreported because they were false positives.
3. *For each old diagnostic, evaluate where it would occur in the new codebase.* This process involves examining the differences produced in Step 1, and seeing how they would apply to the path name and line number where each diagnostic occurs. Some diagnostics might not appear at all in the new codebase if their corresponding file has been removed, or the source code containing their line number has been deleted. For this step, we will assume if the line of code containing the diagnostic has been modified, the diagnostic no longer applies and can be removed. But if the line of code still exists, even though it may have moved in the source file, the diagnostic is still hypothetically possible, and should be preserved, albeit with a new line number. The result of this process should be a list of ‘hypothetical’ diagnostics. They may or may not actually exist, but they refer to valid lines in the new codebase.
4. *For each hypothetical diagnostic in the new codebase, determine if it was automatically generated by the SCALe tools, and if so, copy any information regarding the diagnostic’s validity to the set of diagnostics for the new codebase.* This step is accomplished by determining if a hypothetical diagnostic and a real diagnostic share the same path name, line number, and checker. We can assume that if two diagnostics share this info, they refer to the same issue and their information can be shared.

All of these steps are automated by SCALe scripts. This procedure serves to optimize the manual analysis of the diagnostics produced by the SCALe tools. Since manual analysis is the most time-consuming component of the SCALe audit, this procedure provides a significant improvement in performance and reduction of auditor time, and hence cost.

# Procedure

C++ programs can be analyzed by an extensive number of static analysis (SA) tools. Our experience with static analysis tools has led us to the conclusion that each SA tool has its own strengths and weaknesses, and every tool can detect faults undetectable by other tools. The National Security Agency Center for Assured Software (NSA CAS) ran similar experiments with C/C++ and Java static analysis tools (Willis & Britton, 2011), and they came to the same conclusion. Consequently, running only one SA tool is likely to miss many faults that other tools can detect.

We therefore employ a coverage analysis technique, where we use several SA tools to detect vulnerabilities and merge their results. This technique has several advantages; the biggest one being that we minimize the risk of overlooking critical vulnerabilities (that is, false negatives). Because of the different strengths of different tools, we can also gain new perspectives on vulnerabilities identified by multiple analyzers.

Many tools rely on the assumption that it is more prudent for an SA tool, when encountering some questionable code, to report it as a potential vulnerability than to ignore it. This assumption also enables a security analyst to manually inspect the code and confirm the vulnerability or eliminate it. It minimizes the possibility of ‘false negatives’, that is, uncaught vulnerabilities. However, it does increase the number of false positives; that is, code constructs that might be vulnerable, but turn out to be perfectly legitimate when taken in their total context.

Several tools yield many false positives. Validating each of these diagnostics requires an inspection of the code in question, but sometimes it is necessary to inspect the entire method or class containing the code, or all methods that invoke the method containing the questionable code. Consequently, an auditor has no hope of thoroughly inspecting each and every diagnostic that may be generated by an automated SA tool.

## SEI CERT Coding Rules

An essential element of secure coding in any programming language is well-documented and enforceable coding standards. Coding standards encourage programmers to follow a uniform set of rules and guidelines determined by the requirements of the project and organization, rather than by the programmer's familiarity or preference. Once established, these standards can be used as a metric to evaluate source code (using manual or automated processes).

The CERT Division has published *The SEI CERT C++ Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems in C++ (2016 Edition)*. This book provides rules and recommendations for secure coding in the C++ programming language. The goal of these rules and recommendations is to eliminate insecure coding practices. The application of the secure coding standard will lead to higher quality systems that are robust and more resistant to attack. This coding standard affects the wide range of products coded in C, such as PCs, game players, mobile phones, home appliances, and automotive electronics. It is designed specifically for code conforming to C11, with some support for POSIX and Windows. The CERT Division provides certification for code that is conformant with the *SEI CERT C++ Coding Standard*. The standard is available at the following web address:

<https://www.securecoding.cert.org/confluence/x/HQE>

The *SEI CERT C++ Coding Standard* is partially based on the *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition)*. This book provides rules and recommendations for secure coding in the C programming language. Because C++ is based on C, the C++ coding rules leverage the C coding rules, clearly indicating when a C rule is applicable to C++, and when it can be ignored. The *SEI CERT C Coding Standard* is available at the following web address:

<https://www.securecoding.cert.org/confluence/x/HQE>

The CERT C++ standard consists of nearly 100 rules, not counting applicable rules from the CERT C standard. Coding practices are defined to be rules when the following conditions are met:

1. Violation of the coding practice is likely to result in a [security flaw](https://www.securecoding.cert.org/confluence/display/seccode/BB.+Definitions#BB.Definitions-securityflaw) that may result in an exploitable [vulnerability](https://www.securecoding.cert.org/confluence/display/seccode/BB.+Definitions#BB.Definitions-vulnerability).
2. Conformance to the coding practice can be determined through automated analysis, formal methods, or manual inspection techniques.

Implementation of the secure coding rules defined in this standard are necessary (but not sufficient) to ensure the security of software systems developed in the C++ programming language.

To ensure that the source code conforms to this secure coding standard, it is necessary to have measures in place that check for rule violations. The most effective means of achieving this conformance is to use one or more static analysis tools. Where a rule cannot be checked by a tool, then a manual review is required.

The following two figures illustrate a breakdown of the current rules and recommendations provided by both standards.

Figure : Rules for C++

Figure : Rules and Recommendations for C

### Risk Assessment

Each guideline has an assigned priority. Priorities are assigned using a metric based on Failure Mode, Effects, and Criticality Analysis (FMECA) [[IEC 60812](https://www.securecoding.cert.org/confluence/display/seccode/AA.+Bibliography#AA.Bibliography-IEC608122006)]. Three values are assigned for each guideline on a scale of 1 to 3 for

**severity** - how serious are the consequences of the guideline being ignored

1 = low (denial-of-service attack, abnormal termination)

2 = medium (data integrity violation, unintentional information disclosure)

3 = high (run arbitrary code, privilege escalation)

**likelihood** - how likely is it that a [flaw](https://www.securecoding.cert.org/confluence/display/java/BB.+Definitions#BB.Definitions-securityflaw) introduced by ignoring the guideline could lead to an exploitable vulnerability

1 = unlikely

2 = probable

3 = likely

**remediation cost** - how expensive it is to comply with the guideline

1 = high (manual detection and correction)

2 = medium (automatic detection and manual correction)

3 = low (automatic detection and correction)

The three values are then multiplied together for each guideline. This product provides a measure that can be used in prioritizing the application of the guidelines. These products range from 1 to 27. Guidelines with a priority in the range of 1-4 are level 3 guidelines, 6-9 are level 2, and 12-27 are level 1. As a result, it is possible to claim level 1, level 2, or complete compliance (level 3) with a standard by implementing all guidelines in a level, as shown in Figure 5.

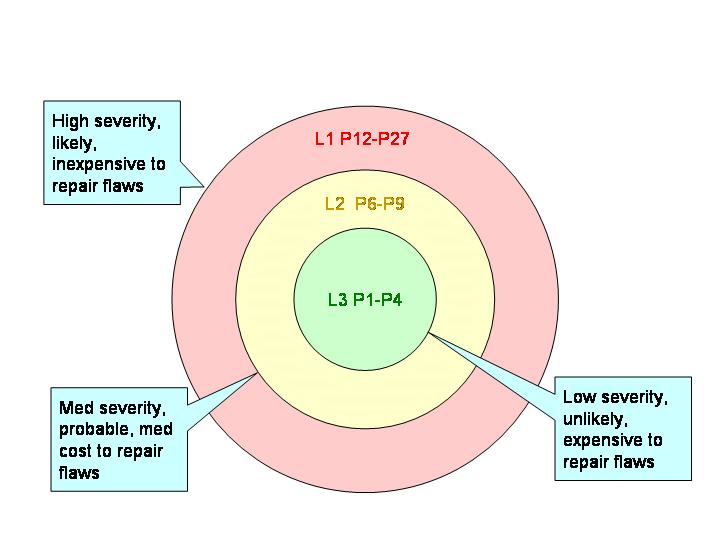


Figure : CERT Secure Coding Priority and Levels

## Diagnostic Categorization

Fortunately, many vulnerabilities rely on a relatively small handful of errors in coding technique, and many SA tools rely on a handful of heuristics to identify vulnerabilities. SA tools typically provide their own categorization of diagnostics and often assign a unique identifier for each diagnostic category. Furthermore, the diagnostics produced by SA tools can be easily associated with CERT rules, where a valid diagnostic indicates a violation of the associated CERT rule. While our SA tools produced many diagnostics, these diagnostics could be classified into violations of a few CERT rules.

Therefore, our approach involves collecting all diagnostics produced by all of the SA tools at our disposal and classifying them by the secure coding guideline they can be associated with. For each secure coding guideline, we then examine a handful of diagnostics. Any diagnostic that turns out to be a false positive is removed immediately. Any diagnostic that turns out to be a *true positive*, that is, indicates a true vulnerability in the code, is added to a table of confirmed vulnerabilities. We examine diagnostics for each guideline until we exhaust all of the diagnostics for that guideline, or until we have found a true positive. Using this process, we can produce a very small set of representative confirmed diagnostics, plus a large set of unconfirmed diagnostics. For every unconfirmed diagnostic, there exists at least one confirmed diagnostic with the same properties.

Some diagnostics are labeled ‘suspicious’. This label indicates that a diagnostic might conceivably be True under certain circumstances, but it was not inspected by an auditor. These uninspected portions of code may or may not be vulnerable to exploits. The code might actually be safe but difficult to analyze. It might actually be safe to use in certain limited contexts, and unsafe in others.

In any case, the code merits attention, and should probably be modified (depending on an organization’s priorities, funding, and effort availability). Without a fix, it is likely that the code may be passed to a maintainer who fails to understand the code and makes incorrect assumptions about its security. Incorrect assumptions increase the maintenance costs of the code, as the maintainer might modify it unnecessarily or might use it improperly, creating one or more new vulnerabilities.

This report provides the complete table of confirmed diagnostics, providing details associated with each. It also provides a similar table of suspicious diagnostics.

## Static Analysis Tools

We have employed the following SA tools, as described below:

### MSVC /analyze

Several editions of Microsoft Visual C++ provide a built-in static analysis tool. This tool includes MSVC 2008 Team Edition, and several editions of MSVC 2010. It is named *analyze mode* because of the /analyze option that is fed to the Microsoft C++ compiler commad. This tool can be enabled by turning on a switch called *Enable Code Analysis*. Consequently, any C/C++ program compiled by Visual Studio will be examined by the SA tool.

### PC-Lint

PC-Lint is a commercial static-analysis tool produced by Gimpel Software for the C and C++ programming languages. First released in 1985, it is supported on all versions of Windows, as well as MS-DOS and OS/2. It provides a command-line interface, but can also be integrated into many IDEs as an external tool, including Microsoft Visual Studio. It provides references to several coding guidelines such as MISRA-C (both 2004 and 2008 editions). More information on PC-Lint is available at the following link:

<http://www.gimpel.com>

### Fortify 360 SCA

Fortify 360 is a commercial product developed by Fortify Software. The product provides an extensive suite of tools for software security assurance. We focused on the source code analysis (SCA) tool. It can be used to analyze software written in C; C++; Java; .NET; ASP.NET; ColdFusion; "Classic" ASP, PHP, VB6, VBScript, JavaScript, PL/SQL, T-SQL and COBOL; as well as configuration files. More information on Fortify Software is available at the following link:

<http://www.fortify.com>

### Coverity Prevent

Coverity Prevent TM is a commercial product developed by Coverity, Inc. The product also provides an extensive suite of tools for software security assurance. We focused on the Coverity Static Analysis tool, which can be used to analyze software written in C, C++, Java, or C#. We also utilized the Coverity Integrity Manager, a web-based framework for viewing the results of Coverity Static Analysis. It provides a rich detail of each diagnostic found, including multiple locations in the source code that serve to create the diagnostic. More information on Coverity is available at the following link:

<http://www.coverity.com>

### Rosecheckers

The Rosecheckers project has been internally developed at the CERT Division to provide a static analysis tool for analyzing C and C++ code. The project was designed to enforce the rules in the *CERT C Secure Coding Standard*. Each rule in the standard that can be statically analyzed has one or more code checkers as part of the Rosecheckers project. The source for the Rosecheckers project is freely downloadable at the following link:

[http://rosecheckers.sourceforge.net](http://rosecheckers.sourceforge.netwww.coverity.com)

The website also provides a virtual machine containing a complete build of the Rosecheckers project on Linux.

The Rosecheckers project leverages the Compass/ROSE project developed at Lawrence Livermore National Labs. This project provides a high-level API for accessing the abstract syntax tree (AST) of a C or C++ source code file. More information on Compass/ROSE is available at the following link:

<http://rosecompiler.org>

### Cppcheck

Cppcheck is a free static analysis tool for C and C++.  More information about cppcheck is available at the following link:

<http://cppcheck.sourceforge.net/>

### Other Tools

Most compilers provide warnings for questionable code. Consequently, a compiler can serve as a simple SA tool, although compilers provide significantly fewer diagnostics than dedicated tools. Furthermore, several SA tools require the software to be compiled in order to function. Coverity, for instance, operates by monitoring a build as it progresses, and running its analysis on each file as it is compiled. Consequently, a program that cannot be completely built cannot be completely analyzed by Coverity.

Because of this, compilation of the software is a crucial first step, and we harvest any diagnostics produced by the compiler and perform the same analysis on them as we do for other SA tools.

Finally, auditors have the ability to provide diagnostics manually, if no tool reports them. It should be emphasized that manual inspection was not a primary procedure in this analysis; it was performed for two purposes: (1) to gain an intuitive overview of the code, and (2) to validate diagnostics produced by the SA tools. Nonetheless, a few diagnostics were might be noted during manual inspection; hence they are included in this report.

## History

A [SCALe](http://www.cert.org/secure-coding/products-services/scale.cfm) audit is a component of quality assurance for a codebase. It is often useful for a codebase to undergo an iterative process of SCALe audits and diagnostic mitigations; this is usually necessary for the codebase to comply with a CERT secure coding standard. Consequently, a codebase might be analyzed with SCALe multiple times. Sometimes the diagnostics reported in a previous audit may remain unfixed for various technical or business reasons. Furthermore, the SCALe process does not report to clients any diagnostics that are known to be false positives, and consequently code that produces such diagnostics is not modified, causing the false diagnostics to recur in subsequent audits.

For any codebase, a SCALe audit provides results that are significant and useful for any future audit of the codebase. When conducting an audit of any codebase that has undergone a previous audit, we can use the list of diagnostics from the previous audit, including any diagnostics that were discovered to be false positives and not presented to the client. If a diagnostic appears in a previous audit and was studied, then the information learned during the previous audit serves as a hint as to the diagnostic’s validity in the current audit. For example, a diagnostic that was revealed to indicate a true vulnerability in a previous audit is likely to still be true in the current audit. It would not be prudent to judge it true automatically. But if it is in a list of a hundred diagnostics, and was revealed to be true in a previous audit, we could choose to examine it first in the current audit. If it is still true, we could mark it so and proceed to the next batch of diagnostics.

SCALe uses a procedure to cascade diagnostic information from a previous analysis into a current analysis. The procedure requires the previous SCALe analysis results, the automatically-generated diagnostics for the new codebase, and the source code for both the old and current versions of the codebase. The procedure is as follows:

1. *Compute the differences between the old and new codebases.* This is easily accomplished using the UNIX diff(1) command. It might require some renaming of files. For example, if the old codebase has a directory named src-1.1, but the new codebase has the same directory named src-1.2, then the filename differences should be resolved. We are less interested in files that have been added, deleted, or moved around in the codebase, and we are more interested in source code files whose lines have been modified.
2. *Gather the old diagnostics, including false positives.* This may involve identifying diagnostics that were unreported because they were false positives.
3. *For each old diagnostic, evaluate where it would occur in the new codebase.* This process involves examining the differences produced in Step 1, and seeing how they would apply to the path name and line number where each diagnostic occurs. Some diagnostics might not appear at all in the new codebase if their corresponding file has been removed, or the source code containing their line number has been deleted. For this step, we will assume if the line of code containing the diagnostic has been modified, the diagnostic no longer applies and can be removed. But if the line of code still exists, even though it may have moved in the source file, the diagnostic is still hypothetically possible, and should be preserved, albeit with a new line number. The result of this process should be a list of ‘hypothetical’ diagnostics. They may or may not actually exist, but they refer to valid lines in the new codebase.
4. *For each hypothetical diagnostic in the new codebase, determine if it was automatically generated by the SCALe tools, and if so, copy any information regarding the diagnostic’s validity to the set of diagnostics for the new codebase.* This step is accomplished by determining if a hypothetical diagnostic and a real diagnostic share the same path name, line number, and checker. We can assume that if two diagnostics share this info, they refer to the same issue and their information can be shared.

All of these steps are automated by SCALe scripts. This procedure serves to optimize the manual analysis of the diagnostics produced by the SCALe tools. Since manual analysis is the most time-consuming component of the SCALe audit, this procedure provides a significant improvement in performance and reduction of auditor time, and hence cost.

# Procedure

The field of static analysis (SA) of Java code is considerably younger than the corresponding field for C or C++, primarily because Java is a younger language. Our experience with static analysis tools has led us to conclude that each SA tool has its own strengths and weaknesses, and every tool can detect faults that are undetectable by other tools. The National Security Agency Center for Assured Software (NSA CAS) ran similar experiments with C/C++ and Java static analysis tools (Willis & Britton, 2011), and they came to the same conclusion. Consequently, running only one SA tool is likely to miss many faults that other tools can detect.

We therefore utilize a coverage analysis technique, in which we employ several SA tools to detect vulnerabilities, and then merge their results. The biggest advantage of this approach is that we minimize the risk of overlooking critical vulnerabilities, also known as false negatives which are uncaught vulnerabilities. Different tools have different strengths, providing us new perspectives on any given vulnerability that is identified by any given analyzer.

Many tools rely on the assumption that it is more prudent for an SA tool, when encountering some questionable code, to report it as a potential vulnerability than for the tool to ignore it. This also enables a security analyst to manually inspect the code and confirm the vulnerability or eliminate it. It minimizes the possibility of false negatives. However, it does increase the number of false positives—that is, code constructs that might be vulnerable but turn out to be perfectly legitimate when taken in their total context.

Several tools yield many false positives. Validating each of these diagnostics requires an inspection of the code in question, but sometimes it is necessary to inspect the entire method or class containing the code, or all methods that invoke the method containing the questionable code. Consequently, an auditor has no hope of thoroughly inspecting each and every diagnostic that may be generated by an automated SA tool.

## SEI CERT Coding Rules

An essential element of secure coding in any programming language is well-documented and enforceable coding standards. Coding standards encourage programmers to follow a uniform set of rules and guidelines determined by the requirements of the project and organization rather than by the programmer’s familiarity or preference. Once established, these standards can be used as a metric to evaluate source code by using manual or automated processes.

The CERT Division has published *The CERT® Oracle® Secure Coding Standard for Java™*. This book provides rules and recommendations for secure coding in the Java programming language. The goal of these rules and recommendations is to eliminate insecure coding practices. The application of the secure coding standard leads to higher-quality systems that are robust and also more resistant to attack. This coding standard affects the wide range of products coded in Java, such as PCs, game players, mobile phones, home appliances, and automotive electronics. It is designed specifically for code conforming to Java 7, with some support for Android. The CERT Division provides certification for code that is conformant with the *CERT Oracle Secure Coding Standard for Java*. The standard is available at the following web address:

<https://www.securecoding.cert.org/confluence/display/java>

This standard consists of over 200 rules and recommendations. Coding practices are defined to be rules when the following conditions are met:

1. Violation of the coding practice is likely to cause a [security flaw](https://www.securecoding.cert.org/confluence/display/seccode/BB.+Definitions#BB.Definitions-securityflaw) that may result in an exploitable [vulnerability](https://www.securecoding.cert.org/confluence/display/seccode/BB.+Definitions#BB.Definitions-vulnerability).
2. Conformance to the coding practice can be determined through automated analysis, formal methods, or manual inspection techniques.

Implementation of the secure coding rules defined in this standard are necessary (but not sufficient) to ensure the robustness and security of software systems developed in the Java programming language.

Recommendations are guidelines or suggestions. Coding practices are defined to be recommendations when all of the following conditions are met:

1. Application of the coding practice is likely to improve system security.
2. One or more of the requirements necessary for a coding practice to be considered a rule cannot be met.

The set of recommendations that a particular development effort adopts depends on the security requirements of the final software product. Projects with high-security requirements can dedicate more resources to security and are consequently likely to adopt a larger set of recommendations.

To ensure that the source code conforms to this secure coding standard, measures that check for rule violations must be provided. The most effective means of checking the source code is to use one or more SA tools. When a rule cannot be checked by a tool, a manual review is required.

Figure 9 details a breakdown of the current guidelines provided by the standard. This lists the sections in the *CERT Oracle Secure Coding Standard for Java*. Each section is associated with the number of guidelines in that section.

Figure : Rules and Recommendations for Java

### Risk Assessment

Each guideline has an assigned priority. Priorities are assigned using a metric based on Failure Mode, Effects, and Criticality Analysis (FMECA) [[IEC 60812](https://www.securecoding.cert.org/confluence/display/seccode/AA.+Bibliography#AA.Bibliography-IEC608122006)]. Three values are assigned for each guideline, on a scale of 1 to 3, for the following:

* **Severity**—How serious are the consequences of the guideline being ignored?  
  1 = Low (denial-of-service attack, abnormal termination)  
  2 = Medium (data integrity violation, unintentional information disclosure)  
  3 = High (run arbitrary code, privilege escalation)
* **Likelihood**—How likely is it that a [flaw](https://www.securecoding.cert.org/confluence/display/java/BB.+Definitions#BB.Definitions-securityflaw) introduced by ignoring the guideline could lead to an exploitable vulnerability?  
  1 = Unlikely  
  2 = Probable  
  3 = Likely
* **Remediation cost**—How expensive is it to comply with the guideline?  
  1 = High (manual detection and correction)  
  2 = Medium (automatic detection and manual correction)  
  3 = Low (automatic detection and correction)

The three values are then multiplied together for each guideline. This product provides a measure that can be used in prioritizing the application of the guidelines. The products range from 1 to 27. Guidelines with a priority in the range of 1 to 4 are level 3 guidelines, 6 to 9 are level 2, and 12 to 27 are level 1. As a result, it is possible to claim level 1, level 2, or complete compliance, level 3, with a standard by implementing all guidelines in a level, as shown in Figure 7.

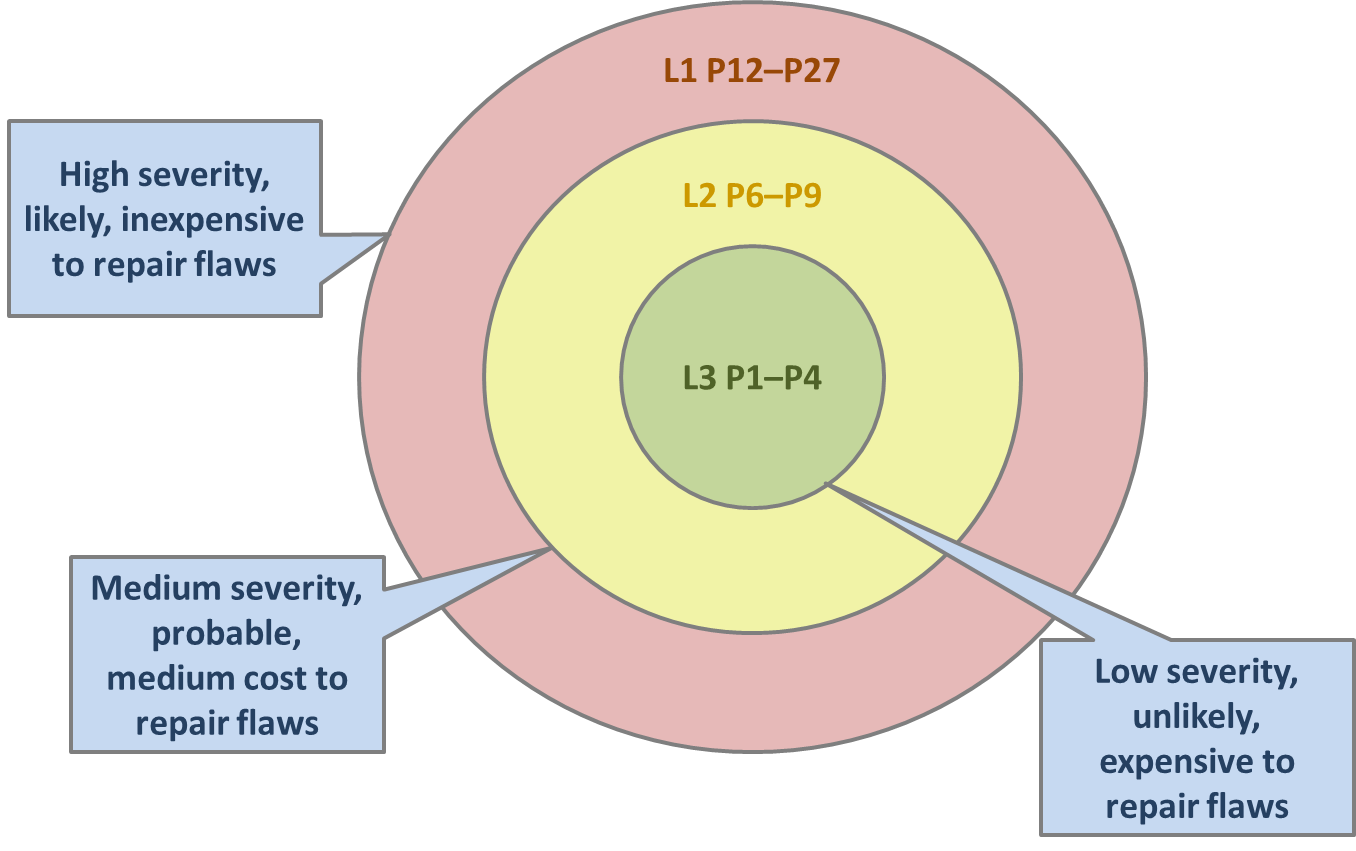


Figure : CERT Secure Coding Priority and Levels

## Diagnostic Categorization

Fortunately, many vulnerabilities rely on a relatively small handful of errors in coding technique, and many SA tools rely on a handful of heuristics to identify vulnerabilities. SA tools typically provide their own categorization of diagnostics and often assign a unique identifier for each diagnostic category. Furthermore, the diagnostics produced by SA tools can be easily associated with CERT rules, where a valid diagnostic indicates a violation of the associated CERT rule. While our SA tools produced many diagnostics, these diagnostics could be classified into violations of a few CERT rules.

Therefore, our approach involves collecting all diagnostics produced by all of the SA tools at our disposal and classifying them by the secure coding guideline with which they can be associated. For each secure coding guideline, we examine a handful of diagnostics. Any diagnostic that turns out to be a false positive is removed immediately. Any diagnostic that turns out to be a *true positive*, one that indicates a true vulnerability in the code, is added to a table of confirmed vulnerabilities. We examine diagnostics for each guideline until we exhaust all of the diagnostics for that guideline or until we have found a true positive. By this process, we can produce a very small set of representative confirmed diagnostics plus a large set of unconfirmed diagnostics. For every unconfirmed diagnostic, there exists at least one confirmed diagnostic with the same properties.

Some diagnostics are labeled “Suspicious,” indicating that a diagnostic might conceivably be true under certain circumstances but was not inspected by any auditor. These diagnostics indicate portions of code that may or may not be vulnerable to exploits. The code might actually be safe but difficult to analyze. It might be safe to use in certain limited contexts and unsafe in others.

In any case, the code merits attention and should probably be modified (depending on an organization’s priorities, funding, and effort availability). Without a fix, it is likely that the code may be passed to a maintainer who fails to understand the code and makes incorrect assumptions about its security. Incorrect assumptions increase the maintenance costs of the code, as the maintainer might modify it unnecessarily or might use it improperly, creating one or more new vulnerabilities. This report provides the complete table of confirmed diagnostics, including details associated with each. It also provides a similar table of suspicious diagnostics.

## Static Analysis Tools

We have employed the following SA tools, as described below:

### FindBugs

FindBugs is an open source program that looks for bugs in Java code. It uses static analysis to identify hundreds of different potential types of errors in Java programs. FindBugs is written in “pure” Java and is therefore platform independent. It provides a simple command-line interface, as well as a graphical interface, and it provides a plug-in enabling it to be integrated into Eclipse.

FindBugs operates on Java bytecode; therefore, it is technically a static binary code analyzer rather than a static source code analyzer. However, it produces useful diagnostics, is free, and thus it was included in our study. For more information on FindBugs, see the following link:

http://findbugs.sourceforge.net

### Fortify 360 SCA

Fortify 360TM is a commercial product developed by Fortify Software. The product itself provides an extensive suite of tools for software security assurance. We focused on the source code analysis (SCA) tool. It can be used to analyze software written in Java, as well as C, C++, .NET, ASP.NET, ColdFusion, “Classic” ASP, PHP, VB6, VBScript, JavaScript, PL/SQL, T-SQL, and COBOL as well as configuration files. More information on Fortify Software is available at the following link:

<http://www.fortify.com>

### Coverity Prevent

Coverity Prevent TM is a commercial product developed by Coverity, Inc. The product also provides an extensive suite of tools for software security assurance. We focused on the Coverity Static Analysis tool, which can be used to analyze software written in C, C++, Java, or C#. We also utilized the Coverity Integrity Manager, a Web-based framework for viewing the results of Coverity Static Analysis. It provides a rich detail of each diagnostic found, including multiple locations in the source code that serve to create the diagnostic. More information on Coverity is available at the following link:

<http://www.coverity.com>

### Other Tools

Most compilers provide warnings for questionable code. Consequently, a compiler can serve as a simple SA tool, although compilers provide fewer diagnostics than dedicated tools. Furthermore, several SA tools require the software to be compiled to function. Coverity, for instance, operates by monitoring a build as it progresses and running its analysis on each file as it is compiled. Consequently, a program that cannot be completely built cannot be completely analyzed by Coverity.

Because of this, compilation of the software is a crucial first step, and we harvest any diagnostics produced by the compiler and perform the same analysis on them as we do for other SA tools.

Finally, auditors have the ability to provide diagnostics manually, if no tool reports them. It should be emphasized that manual inspection was not a primary procedure in this analysis; it was performed for two purposes: (1) to gain an intuitive overview of the code, and (2) to validate diagnostics produced by the SA tools. Nonetheless, a few diagnostics were might be noted during manual inspection; hence they are included in this report.

## History

A SCALe audit is a component of quality assurance for a codebase. It is often useful for a codebase to undergo an iterative process of SCALe audits and diagnostic mitigations, and this is usually necessary for the codebase to comply with a CERT secure coding standard. Consequently, a codebase might be submitted to SCALe multiple times. Sometimes a diagnostic reported in a previous audit may remain unfixed for various technical or business reasons. Furthermore, the SCALe process does not report to clients any diagnostics that are known to be false positives. Consequently, code that produces such diagnostics is not modified, causing the false diagnostics to recur in subsequent audits.

For any codebase, a SCALe audit provides results that are significant and useful for any future audit of the codebase. When conducting an audit of any codebase that has undergone a previous audit, we can use the list of diagnostics from the previous audit, including any diagnostics that were discovered to be false positives and not presented to the client. If a diagnostic appears in a previous audit and was studied, the information learned during the previous audit serves as a hint to the diagnostic’s validity in the current audit. For example, a diagnostic that was revealed to indicate a true vulnerability in a previous audit is likely to still be true in the current audit. It would not be prudent to judge it to be automatically true. But if it is in a list of a hundred diagnostics and was revealed to be true in a previous audit, we could choose to examine it first in the current audit, and if it is still true, we could mark it so and proceed to the next batch of diagnostics.

SCALe uses a procedure to cascade diagnostic information from a previous analysis into a current analysis. The procedure requires the previous SCALe analysis results, the automatically generated diagnostics for the new codebase, and the source code for both the old and current versions of the codebase. The procedure is as follows:

1. *Compute the differences between the old and new codebases.* This step is easily accomplished using the UNIX diff(1) command. It might require some renaming of files. For example, if the old codebase has a directory named src-1.1, but the new codebase has the same directory named src-1.2, then the file name differences should be resolved. We are less interested in files that have been added, deleted, or moved around in the codebase. We are more interested in source code files whose lines have been modified.
2. *Gather the old diagnostics, including false positives.* This step may involve identifying diagnostics that were unreported because they were false positives.
3. *For each old diagnostic, evaluate where it would occur in the new codebase.* This process involves examining the differences produced in step 1 and seeing how they would apply to the path name and line number where each diagnostic occurs. Some diagnostics might not appear at all in the new codebase if their corresponding file has been removed or the source code containing their line number has been deleted. For this step, we assume that if the line of code containing the diagnostic has been modified, the diagnostic no longer applies and can be removed. But if the line of code still exists, even though it may have moved in the source file, the diagnostic is still hypothetically possible and should be preserved, albeit with a new line number. The result of this process should be a list of “hypothetical” diagnostics. They may or may not actually exist, but they refer to valid lines in the new codebase.
4. *For each hypothetical diagnostic in the new codebase, determine if it was automatically generated by the SCALe tools, and if so, copy any information regarding the diagnostic’s validity to the set of diagnostics for the new codebase.* This is accomplished by determining if a hypothetical diagnostic and a real diagnostic share the same path name, line number, and checker. We can assume that if two diagnostics share this information, they refer to the same issue, and their information can be shared.

All these steps are automated by SCALe scripts. This procedure serves to optimize the manual analysis of the diagnostics produced by the SCALe tools. Since manual analysis is the most time-consuming component of the SCALe audit, this procedure provides a significant improvement in performance and reduction of auditor time, and hence cost.

# Procedure

The field of static analysis (SA) of Perl code is considerably younger than the corresponding field for C, primarily because Perl is a younger language than C. Our experience with static analysis tools has led us to the conclusion that each SA tool has its own strengths and weaknesses, and every tool can detect faults undetectable by other tools. The National Security Agency Center for Assured Software (NSA CAS) ran similar experiments with C/C++ and Java static analysis tools (Willis & Britton, 2011), and they came to the same conclusion. Consequently, running only one SA tool is likely to miss many faults that other tools can detect.

We therefore employ a coverage analysis technique, where we use several SA tools to detect vulnerabilities and merge their results. This technique has several advantages; the biggest one being that we minimize the risk of overlooking critical vulnerabilities (that is, false negatives). Because of the different strengths of different tools, we can also gain new perspectives on vulnerabilities identified by multiple analyzers.

Many tools rely on the assumption that it is more prudent for an SA tool, when encountering some questionable code, to report it as a potential vulnerability than to ignore it. This assumption also enables a security analyst to manually inspect the code and confirm the vulnerability or eliminate it. It minimizes the possibility of ‘false negatives’, that is, uncaught vulnerabilities. However, it does increase the number of false positives; that is, code constructs that might be vulnerable, but turn out to be perfectly legitimate when taken in their total context.

Several tools yield many false positives. Validating each of these diagnostics requires an inspection of the code in question, but sometimes it is necessary to inspect the entire method or class containing the code, or all methods that invoke the method containing the questionable code. Consequently, an auditor has no hope of thoroughly inspecting each and every diagnostic that may be generated by an automated SA tool.

## SEI CERT Coding Rules

An essential element of secure coding in any programming language is well-documented and enforceable coding standards. Coding standards encourage programmers to follow a uniform set of rules and guidelines determined by the requirements of the project and organization, rather than by the programmer's familiarity or preference. Once established, these standards can be used as a metric to evaluate source code (using manual or automated processes).

The CERT Division maintains *The SEI CERT Perl Coding Standard*. This website provides rules and recommendations for secure coding in the Perl programming language. The goal of these rules and recommendations is to eliminate insecure coding practices. The application of the secure coding standard will lead to higher quality systems that are robust and more resistant to attack. The standard is available at the following web address:

<https://www.securecoding.cert.org/confluence/x/H4B6Aw>

This standard consists of nearly 50 rules and recommendations. Coding practices are defined to be rules when the following conditions are met:

1. Violation of the coding practice is likely to result in a [security flaw](https://www.securecoding.cert.org/confluence/display/seccode/BB.+Definitions#BB.Definitions-securityflaw) that may result in an exploitable [vulnerability](https://www.securecoding.cert.org/confluence/display/seccode/BB.+Definitions#BB.Definitions-vulnerability).
2. Conformance to the coding practice can be determined through automated analysis, formal methods, or manual inspection techniques.

Implementation of the secure coding rules defined in this standard are necessary (but not sufficient) to ensure the security of software systems developed in the Perl programming language.

Recommendations are guidelines or suggestions. Coding practices are defined to be recommendations when all of the following conditions are met:

1. Application of the coding practice is likely to improve system security.
2. One or more of the requirements necessary for a coding practice to be considered a rule cannot be met.

The set of recommendations that a particular development effort adopts depends on the security requirements of the final software product. Projects with high-security requirements can dedicate more resources to security and are consequently likely to adopt a larger set of recommendations.

To ensure that the source code conforms to this secure coding standard, it is necessary to have measures in place that check for rule violations. The most effective means of achieving this conformance is to use one or more static analysis tools. Where a rule cannot be checked by a tool, then a manual review is required.

Figure 11 illustrates a breakdown of the current rules and recommendations provided by the standard.

Figure : Rules and Recommendations for Perl

### Risk Assessment

Each guideline has an assigned priority. Priorities are assigned using a metric based on Failure Mode, Effects, and Criticality Analysis (FMECA) [[IEC 60812](https://www.securecoding.cert.org/confluence/display/seccode/AA.+Bibliography#AA.Bibliography-IEC608122006)]. Three values are assigned for each guideline on a scale of 1 to 3 for

**severity** - how serious are the consequences of the guideline being ignored

1 = low (denial-of-service attack, abnormal termination)

2 = medium (data integrity violation, unintentional information disclosure)

3 = high (run arbitrary code, privilege escalation)

**likelihood** - how likely is it that a [flaw](https://www.securecoding.cert.org/confluence/display/java/BB.+Definitions#BB.Definitions-securityflaw) introduced by ignoring the guideline could lead to an exploitable vulnerability

1 = unlikely

2 = probable

3 = likely

**remediation cost** - how expensive it is to comply with the guideline

1 = high (manual detection and correction)

2 = medium (automatic detection and manual correction)

3 = low (automatic detection and correction)

The three values are then multiplied together for each guideline. This product provides a measure that can be used in prioritizing the application of the guidelines. These products range from 1 to 27. Guidelines with a priority in the range of 1-4 are level 3 guidelines, 6-9 are level 2, and 12-27 are level 1. As a result, it is possible to claim level 1, level 2, or complete compliance (level 3) with a standard by implementing all guidelines in a level, as shown in Figure 5.

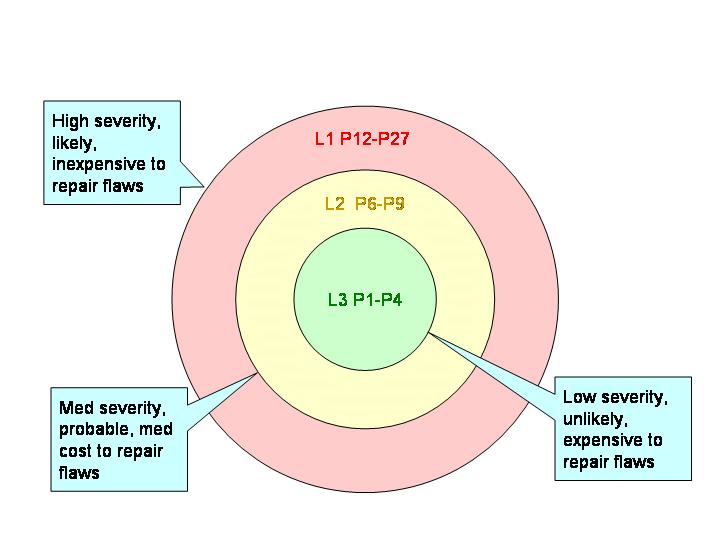


Figure : CERT Secure Coding Priority and Levels

## Diagnostic Categorization

Fortunately, many vulnerabilities rely on a relatively small handful of errors in coding technique, and many SA tools rely on a handful of heuristics to identify vulnerabilities. SA tools typically provide their own categorization of diagnostics and often assign a unique identifier for each diagnostic category. Furthermore, the diagnostics produced by SA tools can be easily associated with CERT rules, where a valid diagnostic indicates a violation of the associated CERT rule. While our SA tools produced many diagnostics, these diagnostics could be classified into violations of a few CERT rules.

Therefore, our approach involves collecting all diagnostics produced by all of the SA tools at our disposal and classifying them by the secure coding guideline they can be associated with. For each secure coding guideline, we then examine a handful of diagnostics. Any diagnostic that turns out to be a false positive is removed immediately. Any diagnostic that turns out to be a *true positive*, that is, indicates a true vulnerability in the code, is added to a table of confirmed vulnerabilities. We examine diagnostics for each guideline until we exhaust all of the diagnostics for that guideline, or until we have found a true positive. Using this process, we can produce a very small set of representative confirmed diagnostics, plus a large set of unconfirmed diagnostics. For every unconfirmed diagnostic, there exists at least one confirmed diagnostic with the same properties.

Some diagnostics are labeled ‘suspicious’. This label indicates that a diagnostic might conceivably be True under certain circumstances, but it was not inspected by an auditor. These uninspected portions of code may or may not be vulnerable to exploits. The code might actually be safe but difficult to analyze. It might actually be safe to use in certain limited contexts, and unsafe in others.

In any case, the code merits attention, and should probably be modified (depending on an organization’s priorities, funding, and effort availability). Without a fix, it is likely that the code may be passed to a maintainer who fails to understand the code and makes incorrect assumptions about its security. Incorrect assumptions increase the maintenance costs of the code, as the maintainer might modify it unnecessarily or might use it improperly, creating one or more new vulnerabilities.

This report provides the complete table of confirmed diagnostics, providing details associated with each. It also provides a similar table of suspicious diagnostics.

## Static Analysis Tools

We have employed the following SA tools, as described below:

### Perl::Critic

Perl::Critic is freely downloadable software, using the same license as Perl itself. It provides comprehensive static analysis of Perl code, with 5 levels of severity. Each diagnostic provides an explanation of several paragraphs, complete with compliant and non-compliant code examples. In addition, each diagnostic also provides a reference to the section in *[Conway 2005]* that addresses the rule enforced by the diagnostic. Perl::Critic is hosted at the following link:

<http://www.perlcritic.org>

This website also provides a link to the CPAN site that supports Perl::Critic. This site provides a breakdown of every module component in Perl::Critic that checks for a specific category of vulnerability. These components live in the Perl::Critic package, and are categorized into subpackages like BuiltinFunctions, and InputOutput. Each package has its own page complete with API documentation, as well as a brief summary of what the package checks for, and why. Thus the Perl::Critic CPAN site serves as a handy reference for each category of diagnostics.

We assume the checker for every diagnostic is the module in Perl::Critic that produces the error, and thus we include the module package name (without the leading Perl::Critic package prefix).

### B::Lint

B::Lint is a free static analysis tool, aimed to be the Perl analogue to C’s famous lint static analysis tool. It is part of CPAN, and is freely downloadable. CPAN also provides documentation and hosting for B::Lint at the following link:

http://perldoc.perl.org/B/Lint.html

Unfortunately, B::Lint provides no indication of checkers. Consequently, we match every diagnostic message against a list of regular expressions, and we provide the expression that matched as the checker.

### Other Tools

Auditors have the ability to provide diagnostics manually, if no tool reports them. It should be emphasized that manual inspection was not a primary procedure in this analysis; it was performed for two purposes: (1) to gain an intuitive overview of the code, and (2) to validate diagnostics produced by the SA tools. Nonetheless, a few diagnostics were might be noted during manual inspection; hence they are included in this report.

## History

A [SCALe](http://www.cert.org/secure-coding/products-services/scale.cfm) audit is a component of quality assurance for a codebase. It is often useful for a codebase to undergo an iterative process of SCALe audits and diagnostic mitigations; this is usually necessary for the codebase to comply with a CERT secure coding standard. Consequently, a codebase might be analyzed with SCALe multiple times. Sometimes the diagnostics reported in a previous audit may remain unfixed for various technical or business reasons. Furthermore, the SCALe process does not report to clients any diagnostics that are known to be false positives, and consequently code that produces such diagnostics is not modified, causing the false diagnostics to recur in subsequent audits.

For any codebase, a SCALe audit provides results that are significant and useful for any future audit of the codebase. When conducting an audit of any codebase that has undergone a previous audit, we can use the list of diagnostics from the previous audit, including any diagnostics that were discovered to be false positives and not presented to the client. If a diagnostic appears in a previous audit and was studied, then the information learned during the previous audit serves as a hint as to the diagnostic’s validity in the current audit. For example, a diagnostic that was revealed to indicate a true vulnerability in a previous audit is likely to still be true in the current audit. It would not be prudent to judge it true automatically. But if it is in a list of a hundred diagnostics, and was revealed to be true in a previous audit, we could choose to examine it first in the current audit. If it is still true, we could mark it so and proceed to the next batch of diagnostics.

SCALe uses a procedure to cascade diagnostic information from a previous analysis into a current analysis. The procedure requires the previous SCALe analysis results, the automatically-generated diagnostics for the new codebase, and the source code for both the old and current versions of the codebase. The procedure is as follows:

1. *Compute the differences between the old and new codebases.* This is easily accomplished using the UNIX diff(1) command. It might require some renaming of files. For example, if the old codebase has a directory named src-1.1, but the new codebase has the same directory named src-1.2, then the filename differences should be resolved. We are less interested in files that have been added, deleted, or moved around in the codebase, and we are more interested in source code files whose lines have been modified.
2. *Gather the old diagnostics, including false positives.* This may involve identifying diagnostics that were unreported because they were false positives.
3. *For each old diagnostic, evaluate where it would occur in the new codebase.* This process involves examining the differences produced in Step 1, and seeing how they would apply to the path name and line number where each diagnostic occurs. Some diagnostics might not appear at all in the new codebase if their corresponding file has been removed, or the source code containing their line number has been deleted. For this step, we will assume if the line of code containing the diagnostic has been modified, the diagnostic no longer applies and can be removed. But if the line of code still exists, even though it may have moved in the source file, the diagnostic is still hypothetically possible, and should be preserved, albeit with a new line number. The result of this process should be a list of ‘hypothetical’ diagnostics. They may or may not actually exist, but they refer to valid lines in the new codebase.
4. *For each hypothetical diagnostic in the new codebase, determine if it was automatically generated by the SCALe tools, and if so, copy any information regarding the diagnostic’s validity to the set of diagnostics for the new codebase.* This step is accomplished by determining if a hypothetical diagnostic and a real diagnostic share the same path name, line number, and checker. We can assume that if two diagnostics share this info, they refer to the same issue and their information can be shared.

All of these steps are automated by SCALe scripts. This procedure serves to optimize the manual analysis of the diagnostics produced by the SCALe tools. Since manual analysis is the most time-consuming component of the SCALe audit, this procedure provides a significant improvement in performance and reduction of auditor time, and hence cost.

References

[Conway 2005]

Conway, Damian. *Perl Best Practices: Standards and Styles for Developing Maintainable Code*. (C) 2005, O'Reilly Media, Inc.

[IEC 60812 2006]

International Electrotechnical Commission (IEC). Analysis techniques for system reliability — Procedure for failure mode and effects analysis (FMEA), 2nd ed. (IEC 60812:2006(E)). IEC, 2006.

[Willis 2011]

Chuck Willis and Kris Britton. On Analyzing Static Analysis Tools, National Security Agency Center for Assured Software (NSA CAS), BlackHat, Las Vegas, Nevada, 2011.

1. CERT is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. [↑](#footnote-ref-1)