Software Engineering Institute
Carnegie Mellon University

SCALe Analysis of JasPer Codebase

David Svoboda

April 2015

Secure Coding Initiative
http://www.sei.cmu.edu

Copyright 2015 Carnegie Mellon University

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

This report was prepared for the
SEI Administrative Agent
AFLCMC/PZM
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

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.

This material has been approved for public release and unlimited distribution except as restricted below.

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 marks of Carnegie Mellon University.

DM-0002301

Table of Contents

1 Introduction 1
1.1 Code Overview 1

2 Findings 2
2.1 Future Work 5

3 Analysis of Findings 6
3.1 *Violation:* Null Pointer Dereference 7
3.2 *Violation:* Use of Freed Memory 8
3.3 *Violation:* Small Buffer Overflow 9

4 Diagnostic Findings 10
4.1 Confirmed Diagnostics 10
4.1.1 Checkers 11

5 Procedure 12
5.1 CERT Secure Coding Rules 12
5.1.1 Risk Assessment 15
5.2 Diagnostic Categorization 16
5.3 Static Analysis Tools 17
5.3.1 MSVC /analyze 17
5.3.2 PC-Lint 17
5.3.3 Fortify 360 SCA 17
5.3.4 Coverity Prevent 18
5.3.5 Rosecheckers 18
5.3.6 Other Tools 18
5.4 History 19

References 21

Figures

Figure 1: Violations by Priority 2

Figure 2: Violations by Tool 2

Figure 3: Violations by CERT Rule 3

Figure 4: Rules and Recommendations for C 14

Figure 5: Rules and Recommendations for C++ 15

Figure 6: CERT Secure Coding Priority and Levels 16

Tables

Table 1: Code Size Metrics 1

Table 2: Code Size Metrics Headers 1

Table 3: Audit Summary Statistics 4

Table 4: Audit Summary Statistics Headers 4

Table 5: Diagnostic Column Headers 10

Table 6: Additional Diagnostic Column Headers 10

1 Introduction

The JasPer codebase is an implementation of the JPEG-2000 Part-1 standard (i.e., ISO/IEC 15444-1) [Jasper]. It is built in C for both Linux and Windows, and is offered under a license similar to the MIT license. This report provides information about the software security of the codebase.

For more info, and to freely download JasPer, visit:

http://www.ece.uvic.ca/~frodo/jasper/#download

1.1 Code Overview

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

Table 1: Code Size Metrics

Package	Files	Space	kLoC	ksigLoC	Size
Jasper	61	16.6	34.2	25.1	940

Table 2: Code Size Metrics Headers

Heading	Definition
Files	Number of C files in each module
Space	disk space occupied by each module, in magabytes
kLoC	Lines of source code (/÷1000)
ksigLoC	Lines of significant source code (/÷1000) (without blank lines and comments)
Size	Size, in kilobytes of C source code, ignoring other files, like HTML, properties, etc.

2 Findings

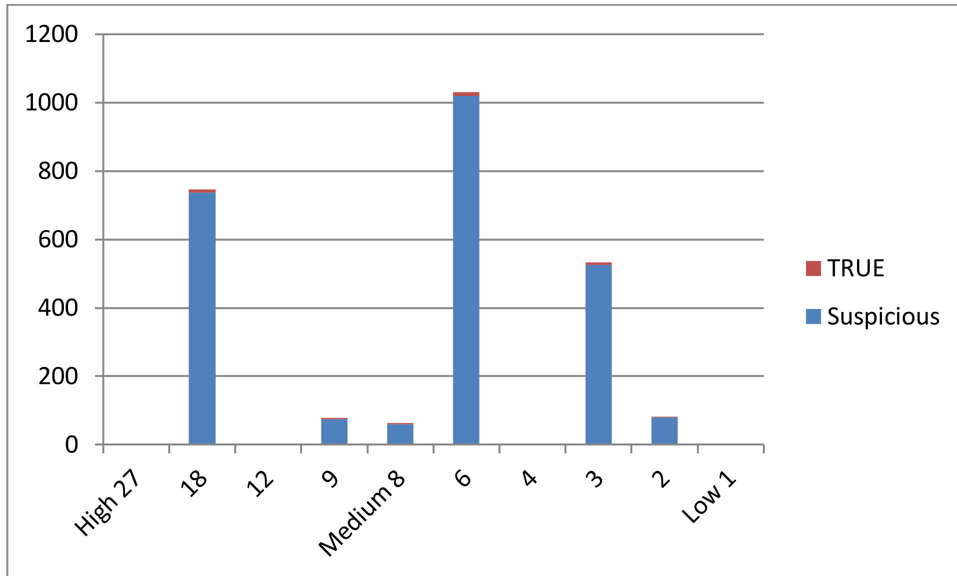


Figure 1: Violations by Priority

Key finding: A few rules of varying priority provided most of the violations.

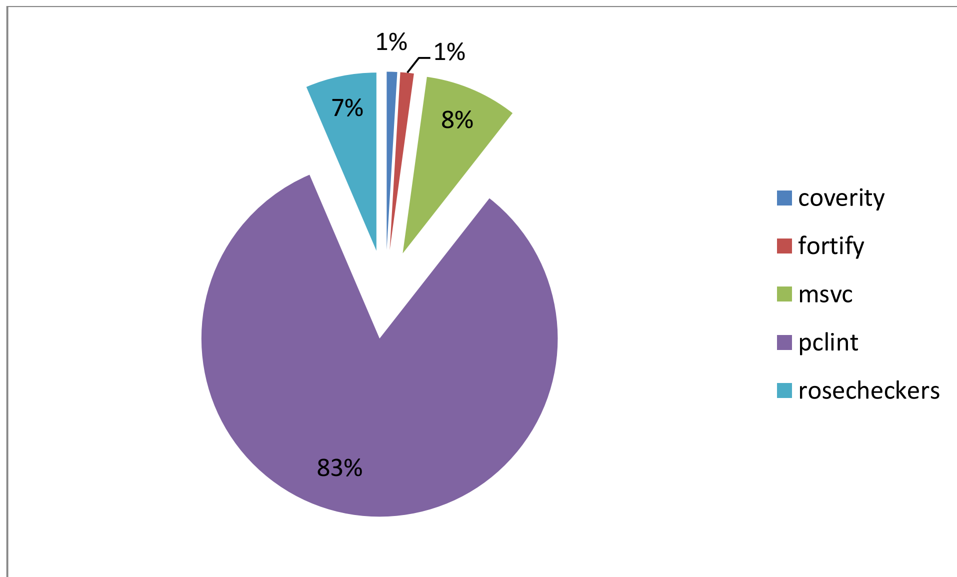


Figure 2: Violations by Tool

Key finding: Most of the tools were helpful in identifying violations. GCC found 8 violations but they were all false positives.

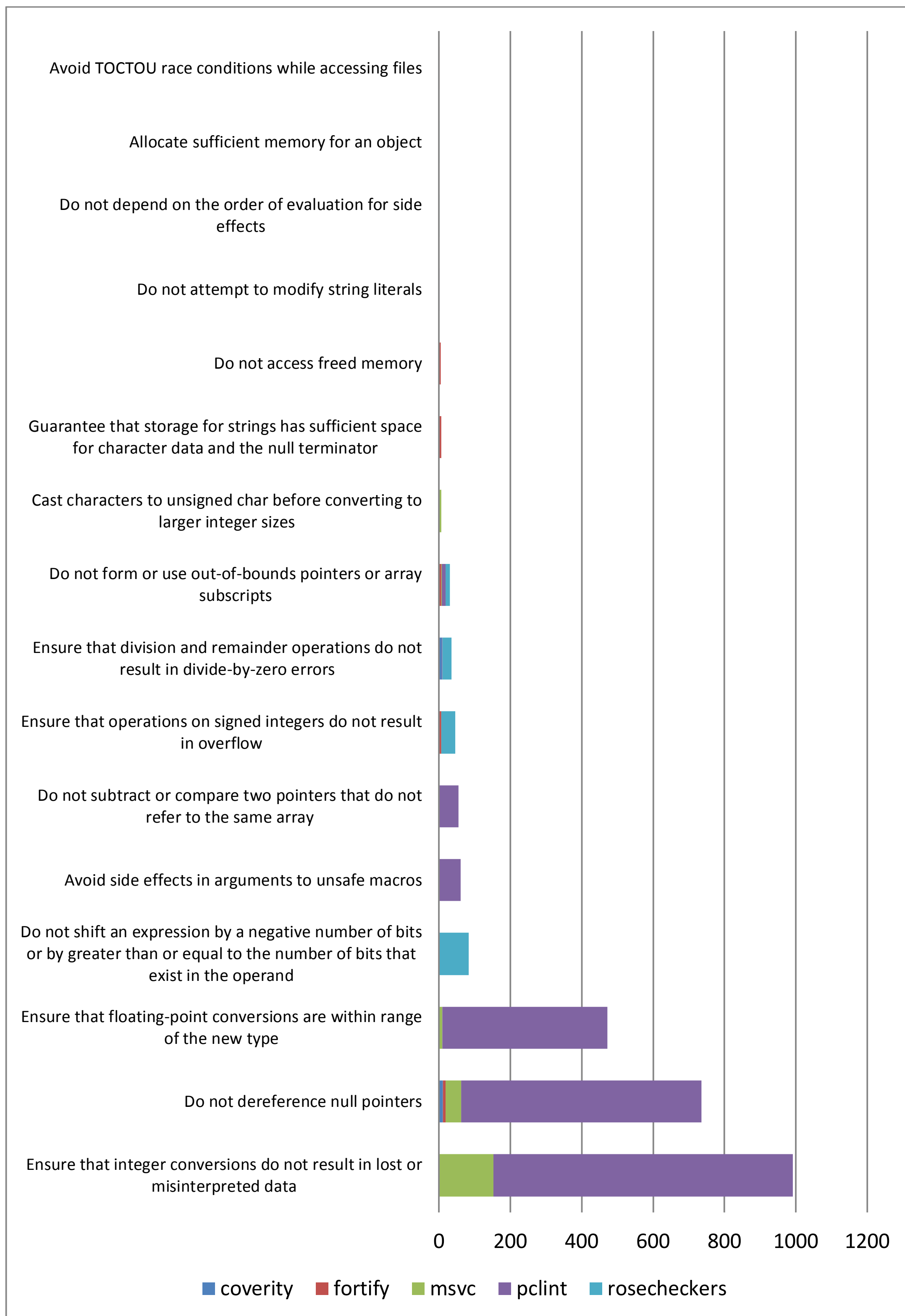


Figure 3: Violations by CERT Rule

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 18, and these violations are the most in need of mitigation, followed by the priority 9 diagnostics, and so forth.

We describe some of the most critical diagnostics in the next section. To comply with the *CERT C Coding Standard*, these diagnostics, along with those in our previous report, must be brought to compliance with the CERT® rules, as described in Section 5.

Several other C/C++ codebases have been audited. Table 3 lists some relevant summary metrics about this codebase in comparison with the others, andTable 4 explains the summary statistics headers.

Table 3: Audit Summary Statistics

	Files	kLoC	ksigLoC	Rules	True	Susp	FileDens	LineDens
Jasper	61	34.2	25.1	16	37	2497	41.5	101
Average	7606	4482.4	3237.1	19.3	99	6202	42	76.1
Std Dev	12516.4	7618.2	5497.5	8.1	54.7	9757.2	71.3	108.8

Table 4: Audit Summary Statistics Headers

Heading	Definition
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: Although this codebase violates fewer rules than average, it has a much higher defect density. The code quality is significantly below average.

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

2.1 Future Work

The spreadsheets 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 C Coding Standard*.

The client, for several reasons, may choose not to modify code that has a 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 diagnostic imposes a constraint on the code that mitigates the violation. This 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 to understand why a diagnostic seems to have been ignored. If the auditor agrees, 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.

3 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. Every rule in the CERT coding standard has a page devoted to 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.

3.1 *Violation:* Null Pointer Dereference

`src/libjasper/base/jas_image.c` has the following code snippet:

```
 208 jas_image_t *jas_image_copy(jas_image_t *image)
 209 {
 210         jas_image_t *newimage;
 211         int cmptno;
 212
 213         newimage = jas_image_create0();
 214         if (jas_image_growcmpts(newimage, image->numcmpts_)) {
 215                 goto error;
 216         }
 217         for (cmptno = 0; cmptno < image->numcmpts_; ++cmptno) {
 218                 if (!(newimage->cmpts_[cmptno] = jas_image_cmpt_copy(image-
>cmpts_[cmptno]))) {
 219                         goto error;
 220                 }
 221                 ++newimage->numcmpts_;
 222         }
...
```

This code assigns `newimage` the output of the `jas_image_create0()` function, and dreferences this pointer on line 218, without ever checking if the pointer was NULL. But it is possible for `jas_image_create0()` to return NULL. From the same file:

```
 186 jas_image_t *jas_image_create0()
 187 {
 188         jas_image_t *image;
 189
 190         if (!(image = jas_malloc(sizeof(jas_image_t)))) {
 191                 return 0;
 192         }
...
```

Consequently, this code violates CERT rule

EXP34-C. Do not dereference null pointers

Solution: Null Check

The simplest solution is to insert a null check before the dereference on line 218. A more thorough solution might be to implement robust error handling. Having `jas_malloc()` invoke `abort()` rather than returning 0 also prevents null pointer dereference (although this might be too drastic for Jasper's purposes.)

3.2 *Violation:* Use of Freed Memory

Here is some code from `src/libjasper/mif/mif_cod.c`:

```
...
 573         jas_tvparser_destroy(tvp);
 574         if (!cmpt->sampperx || !cmpt->samppery) {
 575                 goto error;
 576         }
 577         if (mif_hdr_addcmpt(hdr, hdr->numcmpts, cmpt)) {
 578                 goto error;
 579         }
 580         return 0;
 581
 582 error:
 583         if (cmpt) {
 584                 mif_cmpt_destroy(cmpt);
 585         }
 586         if (tvp) {
 587                 jas_tvparser_destroy(tvp);
 588         }
 589         return -1;
 590 }
```

The `jas_tvparser_destroy()` function frees the pointer given to it, and it is invoked on line 573 and again on line 587. The return statement on line 580 serves to prevent both calls from occuring on the same value of `tvp`. However, if the if statement on line 574, or the one on line 577 are true, the return statement is not executed, control skips to the error label on line 582, and so both calls to `jas_tvparser_destroy()` are invoked. Referencing a pointer after it has been freed (even if to free it a second time) violates CERT rule

MEM30-C. Do not access freed memory

Solution: Adjust Control Flow

The simplest solution is to move the first call to `jas_tvparser_destroy()`to just before the return statement. This move guarentees that `jas_tvparser_destroy()` is invoked exactly once.

3.3 *Violation:* Small Buffer Overflow

In src/libjasper/jp2/jp2_enc.c, we find the following call:

```
346        sprintf(buf, "%s\n_jp2overhead=%lu\n", (optstr ? optstr : ""),
347          (unsigned long) overhead);
```

Any call to sprint() runs the risk of buffer overflow; in this case, overflow will occur if buf is smaller than optstr, or is less than about 25 characters larger than optstr (to accommodate for the format string characters and the stringification of overhead). Determining the capacity of the buf string is fairly straightforward, it is 4096 according to line 100 in the same file. The potential size of optstr is difficult to ascertain, but its contents are built using the addopt() function declared in line 448 of src/appl/jasper.c. This function takes a maxlen argument and guarantees that optstr is limited to this value. However, in both invocations of addopt(), this parameter is set to the macro OPTSMAX, which turns out to be 4097! This means that optstr might be one more characters longer than buf, and certainly long enough for the sprint() command to overflow buf.

This violates CERT rule

STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator

Solution: Sanitize Input Value

There are several simple solutions to this problem. One is to use the C99 function snprintf(), which takes an additional argument indicating the size of buf and guarantees not to overflow this buffer by truncating the formatted string if necessary. Several other functions, such as asprintf() or sprintf_s(), provide similar protections. Another approach is to truncate optstr if it is too long.

4 Diagnostic Findings

The analysis results are provided by two spreadsheets. The "true" spreadsheet indicates flagged nonconformities that were personally verified by our analysts to be violations of the *CERT C Coding Standard*. The "suspicious" spreadsheet indicates 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. Because of their size, the spreadsheets are provided separate from this document. This section documents their contents.

4.1 Confirmed Diagnostics

Each spreadsheet contains the columns listed in Table 5:

Table 5: Diagnostic Column Headers

Header	Definition
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

Each spreadsheet can contain additional columns, although not all diagnostics use them. When diagnostics do use them, the additional columns represent the descriptions listed in Table 6:

Table 6: Additional Diagnostic Column Headers

Heading	Definition
File	Complete path to the file associated with the message
Line	Line number associated with the message
Message	Secondary message describing the violation

Some diagnostics may have two or more messages and links to the source code. For instance, a checker that warns of the use of an uninitialized variable might provide two links (where a link indicates a path name and line number). The first link would be the location where the variable is declared, and the second link would indicate the location where the variable is read (while never being initialized).

The spreadsheets are sorted from lowest level (L1) to highest level (L3), and the diagnostics that occur earlier in each table are more urgent than the diagnostics that occur later. Consequently, we recommend attending to the diagnostics in the order in which they appear, first mitigating the diagnostics in the true violations table and then the diagnostics in the suspicious table.

4.1.1 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.

5 Procedure

C and C++ can be analyzed by an extensive number of *static analysis* (SA) tools. Our experience with C/C++ 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 NSA has made similar experiments with Java static analysis tools and come 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 employ 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.

5.1 CERT Secure 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 CERT C Coding Standard (2nd ed)***.** 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 C99, with some support for POSIX. The CERT Division provides certification for code that is conformant with the *CERT C Secure Coding Standard***.** The standard is available at the following web address:

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

This standard consists of nearly 300 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 that may result in an exploitable 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:

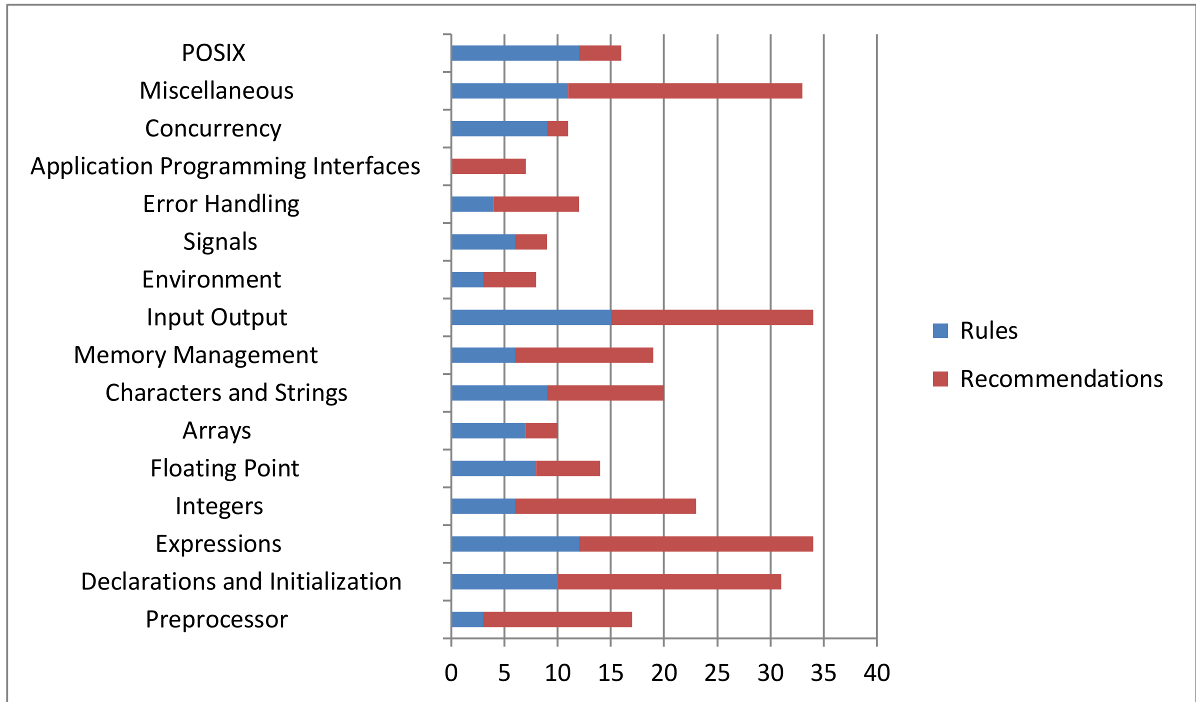
1. Application of the coding practice is likely to improve system security.
2. One or more of the requirements necessary for a coding practice to be considered a rule cannot be met.

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 4: Rules and Recommendations for C



The CERT Division also publishes *The CERT C++ Coding Standard***.** The standard is available at the following web address:

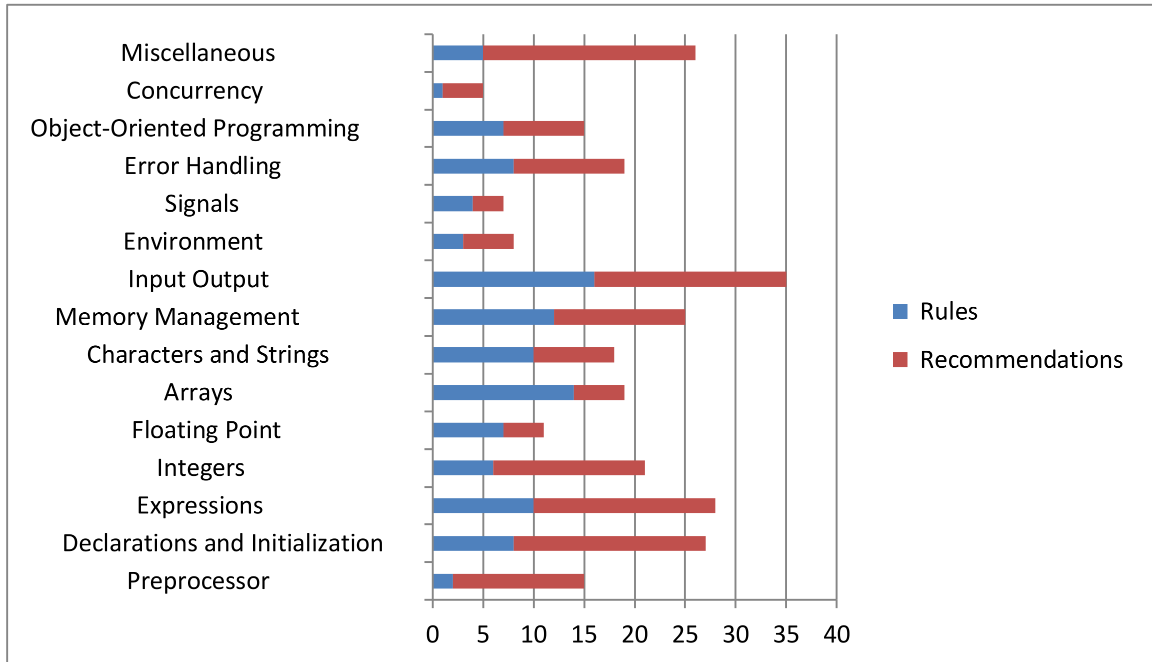
https://www.securecoding.cert.org/confluence/x/fQI

Unlike the C coding standard, *The CERT C++ Coding Standard* is not complete, and the CERT Division does not yet issue certifications for the C++ standard. Consequently, while the SCALe process will sometimes mention adherence to C++ secure coding rules, they are offered on a strictly advisory basis, and our certification is contingent solely on the C rules.

The C++ coding standard is also divided into rules and recommendations, using the same definitions as the C coding standard.

Figure 5 illustrates a breakdown of the current rules and recommendations provided by the standard.

Figure 5: Rules and Recommendations for C++



5.1.1 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]. 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 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 6.

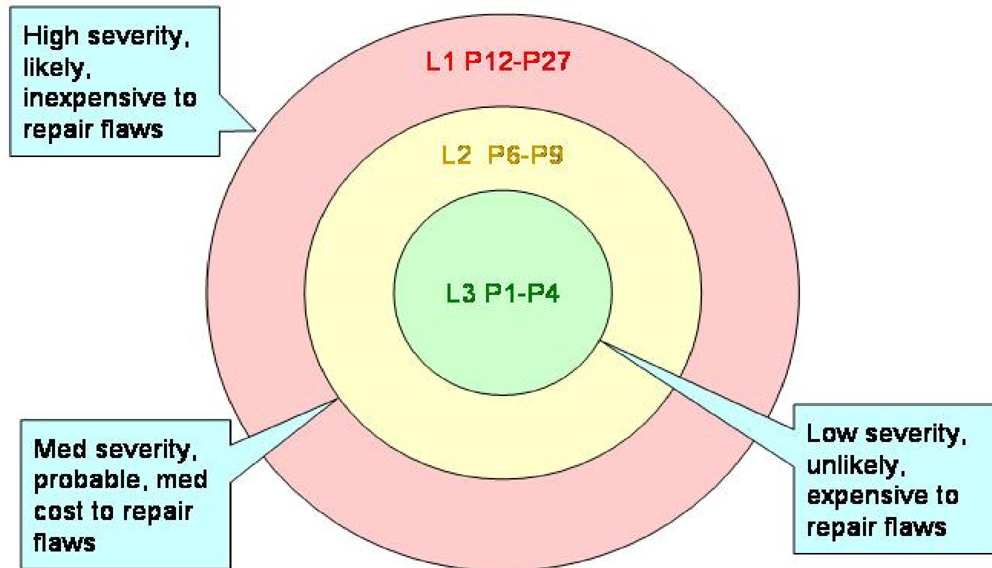


Figure 6: CERT Secure Coding Priority and Levels

5.2 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 secure coding guidelines, where a valid diagnostic indicates a violation of the associated CERT guideline. While our SA tools produced many diagnostics, these diagnostics could be classified into violations of a few secure coding guidelines.

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. 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. Such an occurrence increases 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.

5.3 Static Analysis Tools

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

5.3.1 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.

5.3.2 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

http://www.gimpel.com

5.3.3 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 is available at

http://www.fortify.com

5.3.4 Coverity Prevent

Coverity Prevent 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

http://www.coverity.com

5.3.5 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* and 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

http://rosecheckers.sourceforge.net

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 here

http://rosecompiler.org

5.3.6 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 liability, 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, a few diagnostics were identified by manually inspecting the code. 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 noted during manual inspection; hence they are included in this report.

5.4 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; 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 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 these steps can be automated by scripts. The scripts work well in practice, although they need to be run carefully and checked afterward to ensure that they produced the correct result. Such checking usually involves examining a sample of about three diagnostics to make sure that they were handled correctly by the scripts.

This process serves to optimize the manual analysis of the diagnostics produced by the SCALe tools. Since this is the most time-consuming component of the SCALe audit, this process provides a significant improvement in performance and reduction of auditor time.

References

[Jasper]
The JasPer Project Home Page, http://www.ece.uvic.ca/~frodo/jasper/

[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.