Secure Coding Institute

Research Report

Open Source Software Security Analysis using QA·C

An Empirical Study

This report documents findings from an empirical study in which the Secure Coding Institute analyzed the capabilities of the latest release of PRQA's QA·C static analyzer to detect secure coding violations in an open source software library. PRQA's QA·C analyzer is effective at discovering violations of *The CERT C Coding Standard* that were not discovered through 20 years of testing or by other static analysis tools.

Open Source Software Security Analysis using QA·C

An Empirical Study

Study Overview

In this study, we analyzed the libpng open source library using version 8.2.2 of the QA·C tool from PRQA that includes a compliance module to detect violations of the secure coding rules defined by *The CERT C Coding Standard* [Seacord 2014]. This study documents the specific experiences of analyzing the libpng open source library and evaluates the results of the analysis.

Software Security

The past two years have seen software security evolve from a curiosity to an essential concern as the result of many high profile incidents. The first of these was The Heartbleed bug, a serious vulnerability in the Open SSL cryptographic software library, which enables attackers to steal information that, under normal conditions, is protected by the Secure Socket Layer/Transport Layer Security (SSL/3TLS) encryption used to secure the internet. Heartbleed spurred the creation of The Linux Foundation Core Infrastructure Initiative (CII) project to fund and support free and open-source software projects that are critical to the functioning of the Internet and other major information systems. In April 2015, the United States Office *of Personnel Management* (*OPM*) discovered that the personnel data of 4.2 million current and former Federal government employees had been stolen. While investigating this incident, in early June 2015, OPM discovered that additional information had been compromised: including background investigation records of current, former, and prospective Federal employees and contractors. OPM and the interagency incident response team have concluded with high confidence that sensitive information, including the Social Security Numbers (SSNs) of 21.5 million individuals, was stolen from the background investigation databases. This includes 19.7 million individuals that applied for a background investigation, and 1.8 million non-applicants, primarily spouses or co-habitants of applicants. On July 21, 2015, security researchers demonstrated a remote attack that can be performed against many FCA-(Fiat Chrysler Automobiles) vehicles forcing a 1.4 million vehicle recall as well as changes to the Sprint carrier network. Additionally, the growth of an endless variety of connected devices in the Internet of Things (IoT) poses an equally wide variety of security challenges. As Marc Andreessen famously stated, software (and now software security) is eating the world.

QA·C

• • •

QA·C is a static analysis solution for the C language, providing a comprehensive suite of features to help to enforce a wide range of coding standards, and to find bugs in new and legacy code. QA·C offers an automated analysis of code against the chosen coding standard, with metrics and code structure visualizations.

PRQA

PRQA has been a leader in defect prevention for more than 25 years. Their solutions have been proactively adopted by developers globally, promoting safe coding practices and helping to ensure the highest quality code for safety-critical and mission-critical systems[1].

QA·C

QA·C is a static analysis solution for the C language, providing a comprehensive suite of features to help to enforce a wide range of coding standards, and to find bugs in new and legacy code. QA·C offers an automated analysis of code against the chosen coding standard, with metrics and code structure visualizations. QA·C can be used to prevent bugs and to identify coding issues early in the development cycle. The version 8.2.2 of the QA·C tool evaluated in this study includes a compliance module to detect violations of *The CERT C Coding Standard.*

QA·C ships with a Graphical User Interface tool (`qagui`), a command line interface (`qacli`), and plug-ins for Visual Studio and Eclipse. Only the QA·C Graphical User Interface was used in this study.

The CERT C Coding Standard

The CERT® C Coding Standard, Second Edition, provides rules for coding in the C programming language. The goal of these rules is to develop safe, reliable, and secure systems, for example, by eliminating undefined behaviors that can lead to unexpected program behaviors and exploitable vulnerabilities. Conformance to the coding rules defined in this standard are necessary (but not sufficient) to ensure the safety, reliability, and security of software systems developed in the C programming language. It is also necessary, for example, to have a safe and secure design. Safety-critical systems typically have stricter requirements than are imposed by this coding standard; a well-known case is the request to avoid dynamic memory allocation that can be found in MISRA-C:2012. However, the application of the CERT C Coding Standard will result in high-quality systems that are reliable, robust, and resistant to attack.

libpng

The libpng open source library[1] was selected for analysis. Portable Network Graphics (PNG) is a raster graphics file format that supports lossless data compression [ISO/IEC 2004]. PNG was created as an improved, non-patented replacement for Graphics Interchange Format (GIF), and is the most used lossless image compression format on the Internet. The libpng library is the official PNG reference library. It supports almost all PNG features, is extensible, and has been extensively tested for over 20 years. Regardless of this history, a number of notable vulnerabilities have been discovered in libpng versions including CVE-2014-9495, CVE-2014-0333, CVE-2013-6954, CVE-2012-3386, CVE-2011-3048, and CVE-2011-3026.

[1] http://www.programmingresearch.com/about-us/#sthash.2Dd3RQB2.dpuf

[1] http://www.libpng.org/pub/png/libpng.html

The libpng library is of interest to both the Linux Foundation Core Infrastructure Initiative (CII)[2] and Google's Application Security Patch Reward Program[3]. The CCI Census Project included libpng12-0 in its survey of at risk projects with an overall Risk Index of 7, with 20 CVEs, 9 contributors, and a popularity of 153,194 as maintained by Debian. The CII Census Project website[4] describes how the Risk Index is calculated.

The git repository for libpng can be accessed by cloning git://git.code.sf.net/p/libpng/code or browsed at github.com/glennrp/libpng or at sourceforge.net/p/libpng/code. There are eight branches (libpng00, libpng10, libpng12, libpng14, libpng15, libpng16, libpng17, and a master branch which is roughly equivalent to the libpng16 branch). Development occurs in the libpng16 and libpng17 branches. The Coverity project for libpng[5] is set up to scan the head of each of the libpng10 through libpng17. As of July 23, 2015, libpng-1.0.63, libpng-1.2.53, libpng-1.4.16beta02, libpng-1.5.23, libpng-1.6.18, and libpng-1.7.0beta63 were all reported to be defect-free). The evaluation in this study was performed on the libpng16 branch. All code was compiled using gcc (Ubuntu 4.8.4-2ubuntu1~14.04) 4.8.4 on the Linux Mint Linux Mint 17.2 distribution for the Intel x86-64 architecture.

Completeness and Soundness

It should be recognized that, in general, determining conformance to coding rules is computationally undecidable. The precision of static analysis has practical limitations. For example, the halting theorem of Computer Science states that there are programs whose exact control flow cannot be determined statically. Consequently, any property dependent on control flow—such as halting—may be indeterminate for some programs. A consequence of this undecidability is that it may be impossible for any tool to determine statically whether a given rule which relies on the knowledge of those properties is satisfied in specific circumstances.

However checking is performed, the analysis may generate

— False negatives: Failure to report a real flaw in the code is usually regarded as the most serious analysis error, as it may leave the user with a false sense of security. Most tools err on the side of caution and consequently generate false positives. However, there may be cases where it is deemed better to report some high-risk flaws and miss others than to overwhelm the user with false positives.
— False positives: The tool reports a flaw when one does not exist. False positives may occur because the code is sufficiently complex that the tool cannot perform a complete analysis. The use of features such as function pointers and libraries may make false positives more likely.

To the greatest extent feasible, an analyzer should be both complete and sound with respect to a specific ideal. An analyzer is considered sound with respect to an ideal if it cannot give a false-negative result, meaning it finds all violations of the ideal within the entire program. An analyzer is

[22] https://www.coreinfrastructure.org/
[3] https://www.google.com/about/appsecurity/patch-rewards/index.html
[4] https://www.coreinfrastructure.org/programs/census-project
[5] https://scan.coverity.com/projects/4061

considered complete if it cannot issue false-positive results, or false alarms. The possibilities for a given rule are outlined in the following table.

False positives			
False negatives		Y	N
	N	Sound with false positives	Complete and sound
	Y	Unsound with false positives	Complete and unsound

The ideal against which true and false positives are evaluated depends in part on who is performing the analysis. Common criteria for defining an ideal include:

— Checker: most analyzers are implemented as a collection of checkers which evaluate the code for violations of a particular property. If the property is violated and the analyzer correctly diagnoses this violation it is considered a true positive. This is the approach most commonly used by analyzer vendors to report their results and consequently the approach used in this report for comparability of results.

— Rules: analyzers frequently diagnose violations of rule sets such as those defined by *The CERT® C Coding Standard, Second Edition*, MISRA, or other rule sets. Frequently the mapping between a checker and a rule is inexact. Consequently, measuring completeness and soundness with respect to rules is likely to result in higher false positive and false negative rates.

— Defects: a violation of a checker or a rule does not necessarily indicate an actual defect in the program and may just indicate that the checker or rule is overly strict. This approach is typically used by developers of the code under analysis so it is important from a consumer perspective but may also lead to true positives being mischaracterized because the "code works" or similar reasons.

Analysis

QA·C version 8.2.2 was used to analyze core library files in the libpng16 branch of libpng. Files under the tests directory were not evaluated because of they are not part of the deployed system and were of lower quality. There was not adequate time to analyze all the generated diagnostics, so a sampling was made of interesting *diagnostics*. In this case, "interesting" typically meant diagnostics generated by checkers not yet evaluated. This section is divided into subsections based on the specific CERT C Coding Standard guidelines violated. In many cases, more than one checker will detect violations of a single CERT guideline.

INT13-C. Use bitwise operators only on unsigned operands

The majority of diagnostics apply to expressions of essentially signed type being used as the operand of a bitwise or shift operator. For example, the following code in `png.c` was diagnosed:

```
void PNGAPI png_set_sig_bytes(png_structrp png_ptr, int num_bytes) {
   if (png_ptr == NULL) return;
   if (num_bytes > 8) png_error(png_ptr, "Too many bytes for PNG signature");
   png_ptr->sig_bytes = (png_byte)((num_bytes < 0 ? 0 : num_bytes) & 0xff);
}
                                                                           ^
Msg(qac-8.2.2-4532)  An expression of 'essentially signed' type (signed int) is
being used as the left-hand operand of this bitwise operator (&).
```

This is a violation of the CERT C Coding Recommendation "INT13-C. Use bitwise operators only on unsigned operands". In this case, the `png_set_sig_bytes()` function accepts a `int` argument for `num_bytes` for what should essentially be an unsigned value of type `size_t`. To not change the interface, the following changes were made to the function:

```
void PNGAPI png_set_sig_bytes(png_structrp png_ptr, int num_bytes) {
   unsigned int nb = (unsigned int)num_bytes;
   if (png_ptr == NULL) return;
   if (num_bytes < 0) nb = 0;
   if (nb > 8) png_error(png_ptr, "Too many bytes for PNG signature");
   png_ptr->sig_bytes = (png_byte)nb;
}
```

The solution converts the signed value to an unsigned value and greatly simplifies the original expression, eliminating the diagnostic. There are many examples of similar code being diagnosed with this same violation. Each example examined was a true positive result.

MSC12-C. Detect and remove code that has no effect or is never executed

QA·C checkers 2985, 2992, 2996, and 3112 detects violations of the CERT recommendation MSC12-C. This rule requires code that has no effect or is never executed to be diagnosed. This is important because developer's seldom write code which serves no purpose. Consequently, this is often an indication of defect.

There are a number of diagnostics resulting from debug statements:

```
   png_debug(1, "in png_set_sig_bytes");

      ^
Msg(qac-8.2.2-3112)  This statement has no side-effect - it can be removed.
```

Although uninteresting, these are true positives because the code as analyzed does not define the `png_debug`, `png_debug1`, or `png_debug2` macros and are all consequently expanded to the `((void)0)` statement with no side effect. If the macros are defined using the `-D` option of the parser, for example, these diagnostics will be eliminated. The checker can also be turned off, but that may result in false negatives.

The following diagnostic is a true positive:

```
typedef unsigned int uInt;
typedef size_t png_alloc_size_t;

void png_zalloc (uInt items, uInt size) {
  if (items >= (~(png_alloc_size_t)0)/size) { }
               ^
Msg(qac-8.2.2-2992)  The value of this 'if' controlling expression is always
'false'.
Msg(qac-8.2.2-2996)  The result of this logical operation is always 'false'.

}
```

The `png_alloc_size_t` type is a 64-bit unsigned type while `size` and `items` are 32-bit `unsigned int`. For this implementation, the value of the subexpression `~(png_alloc_size_t)0)` is `0xFFFFFFFFFFFFFFFF` while the value of size can be at most `0xFFFFFFFF`. Consequently, the result of the controlling expression for the `if` statement is always false and the code serves no purpose for this implementation. The code does serve a purpose on implementations where `size_t` is a 32-bit unsigned type. The basic problem with this code is that the size argument to the function is specified as `uInt` and not as `size_t`.

The following is an example of a diagnostic issued by QA·C checkers 2985:

```
void PNGAPI
png_save_int_32(png_bytep buf, png_int_32 i) {
   buf[0] = (png_byte)((i >> 24) & 0xff);
                                  ^
Msg(qac-8.2.2-2985)  This operation is redundant. The value of the result is
always that of the left-hand operand.
-->/src/libpng/png.c:676:43 (qac-8.2.2-1594)    'i' declared here.
```

On an architecture such as the one being evaluated this code is certainly a no-op. However, this code was added for portability for implementations where the number of bits in a byte (`CHAR_BIT`) is larger than 8 bits. Consequently, the diagnostic is a true positive with respect to this implementation and reasonable but a false positive with respect to being a code defect; it's worth to note though that the diagnostic will disappear (as expected) if the configuration will be adapted to an implementation with `stdint.h` declaring `CHAR_BIT >8`.

EXP33-C. Do not read uninitialized memory

The QA·C checker 2971 detects violations of the CERT rule EXP33-C. Uninitialized automatic variables or dynamically allocated memory has indeterminate values, which for objects of some types can be a trap representation. Reading such trap representations is undefined behavior. The following is an example diagnostic:

```
char number_buf[5]; /* enough for a four-digit year */
#define APPEND_STRING(string) pos = png_safecat(out, 29, pos, (string))
#define APPEND_NUMBER(format, value)\
         APPEND_STRING(PNG_FORMAT_NUMBER(number_buf, format, (value)))

APPEND_NUMBER(PNG_NUMBER_FORMAT_u, (unsigned)ptime->day);
            ^
Msg(qac-8.2.2-2971)  Definite: Passing address of uninitialized object
'number_buf' to a function parameter declared as a pointer to const.
-->/src/libpng/png.c:708:12 (qac-8.2.2-1594) 'number_buf' declared here.
```

The actual code is obscured because of macro expansion, but the character array `number_buf` is clearly uninitialized. After macro expansion, we end up with the following call:

```
pos = png_safecat(out, 29, pos, (png_format_number(number_buf, number_buf
+ (sizeof number_buf), 1, ((unsigned)ptime->day))));
```

The png_format_number function has the following signature:

```
png_charp png_format_number(
  png_const_charp start, png_charp end, int format, png_alloc_size_t
number
);
```

As might be expected, png_const_charp is a typedef for const char * which means that the format_number function cannot modify the memory referenced by start. However, the format_number function only modifies the memory referenced by the end pointer and the start pointer is only used to identify the address of the first element in the array. This particular diagnostic is a true positive with respect to the checker, but a false positive with respect to the rule and being a defect.

EXP05-C. Do not cast away a const qualification

The QA·C checker 0311 detects violations of the CERT recommendation "EXP05-C. Do not cast away a const qualification". Casting away the const qualification allows a program to modify the object referred to by the pointer, which may result in undefined behavior. The following error was diagnosed in the png_error function in the pngerror.c file:

```
   if (png_ptr != NULL && png_ptr->error_fn != NULL)
      (*(png_ptr->error_fn))(png_constcast(png_structrp,png_ptr),
                                  ^
Msg(qac-8.2.2-0311)  Dangerous pointer cast results in loss of const
qualification.
          error_message);
```

The png_constcast() is a function like macro that expands to ((png_structrp)png_ptr). The png_ptr argument to the png_error function is declared as a png_const_structrp which is a typedef for const png_struct * while png_structrp is not a const-qualified type. This is a violation of EXP05-C as casting away the const qualification allows a program to modify the object referred to by the pointer, which may result in undefined behavior. A compiler may place a const object that is in a read-only region of storage. Moreover, the implementation need not allocate storage for such an object if its address is never used. This is a true positive finding both in respect to the checker and the rule; resulting in a concrete defect.

EXP34-C. Do not dereference null pointers

The QA·C checker 2822 detects violations of the CERT rule "EXP34-C. Do not dereference null pointers". Dereferencing a null pointer is undefined behavior. Because program termination is not required by the standard, this defect can result in an exploitable vulnerability. The following code was diagnosed by this checker in the png_do_compose function in pngrtran.c:

```
  if (((row_info->bit_depth <= 8 && gamma_table != NULL) ||
        (row_info->bit_depth == 16 && gamma_16_table != NULL)))
                {
        switch (row_info->color_type) {
           case PNG_COLOR_TYPE_RGB: {
              if (row_info->bit_depth == 8) {
                 sp = row;
                 for (i = 0; i < row_width; i++) {
                    *sp = gamma_table[*sp];
                                          ^
Msg(qac-8.2.2-2822)  Apparent: Arithmetic operation on NULL pointer.
```

"Apparent" is a dataflow level diagnostic generated when no further information is available to specify if the pointer is valid; in this case no check is performed on the `row` argument before assigning it to `sp` and then dereferencing it. The `png_do_compose` function is a declared `static` and only accessible from within this compilation unit. The one invocation of the function from within the library does ensure this parameter is not a null pointer.

Conclusions

With the new compliance mode, PRQA's QA·C analyzer is effective at discovering violations of *The CERT C Coding Standard* that were not discovered through 20 years of testing or by other static analysis tools. Because many of the QA·C checkers are heuristic, they can sometimes report true positive findings which do not represent actual defects in the code base. Overall, the QA·C analyzer is an effective tool for eliminating secure coding flaws that can easily lead to software vulnerabilities.

About the Author

ROBERT C. SEACORD is the founder of the Secure Coding Institute which specializes in secure coding consulting and training. The Secure Coding Institute is a trusted provider of operationally

relevant cybersecurity research, secure software development and assessment, and secure coding training. Previously, Robert was the secure coding technical manager in the CERT Division of Carnegie Mellon University's Software Engineering Institute (SEI). Robert is the author of *The CERT® C Coding Standard, Second Edition: 98 Rules for Developing Safe, Reliable, and Secure* Systems, 2nd Edition (Addison-Wesley, 2014) and *Secure Coding in C and C++, Second Edition* (Addison-Wesley, 2013), as well as coauthor of four other books. Robert is an adjunct professor at Carnegie Mellon University and a technical expert for ISO/IEC JTC1/SC22/WG14, the international standardization working group for the programming language C.

References

[ISO/IEC 15948:2004]
International Organization for Standardization, International Electrotechnical Commission (ISO/IEC). *ISO/IEC 15948:2004 Information technology -- Computer graphics and image processing -- Portable Network Graphics (PNG): Functional specification*. ISO, March 2004.

[ISO/IEC 9899:2011]
International Standards Organization. ISO/IEC 9899:2011. *Information Technology — Programming Languages — C*. ISO, 2011.

[ISO/IEC 24731-1:2007]
International Organization for Standardization, International Electrotechnical Commission (ISO/IEC). *ISO/IEC TR 24731-1:2007 Extensions to the C Library — Part I: Bounds-checking interfaces*. ISO, August 2007.

[ISO/IEC TR 24772:2013]
ISO/IEC. *Information Technology—Programming Languages— Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use* (ISO/IEC TR 24772:2013). Geneva, Switzerland: ISO, March 2013.

[ISO/IEC TS 17961:2013]
ISO/IEC. *Information Technology—Programming Languages, Their Environments and System Software Interfaces—C Secure Coding Rules* (ISO/IEC TS 17961). Geneva, Switzerland: ISO, 2012.

[Miller 2015]
Miller, Charlie; Chris Valasek. *Remote Exploitation of an Unaltered Passenger Vehicle*. August 10, 2015.
http://illmatics.com/Remote%20Car%20Hacking.pdf

[MISRA 2012]
Motor Industry Software Reliability Association (MISRA). *MISRA C3: Guidelines for the Use of the C Language in Critical Systems* 2012. Nuneaton, UK: MIRA, 2012.

[Seacord 2014]
Seacord, Robert. *The CERT® C Coding Standard: 98 Rules for Developing Safe, Reliable, and Secure Systems*, 2nd ed. Addison-Wesley Professional, 2014.