ADDRESSING SECURITY VULNERABILITIES IN EMBEDDED APPLICATIONS USING BEST PRACTICE SOFTWARE DEVELOPMENT PROCESSES AND STANDARDS

An introduction to applying CWE coding guidelines and achieving CERT security compliance using static analysis tools
# TABLE OF CONTENTS

SECURITY AND THE NEED FOR STANDARDS AND BEST PRACTICES .................. 3
  Security is a process, not a feature ......................................................... 3

THE SOFTWARE SECURITY PROBLEM:
WHY IT’S SO DIFFICULT TO PRODUCE SECURE SOFTWARE .................... 4
  Security is just not a priority ................................................................. 4
  Quality assurance doesn’t equal security assurance ................................. 4
  Embedded software has too many moving parts ...................................... 4
  Training is inadequate .......................................................................... 4
  Nobody has ownership of security ......................................................... 5

SECURE DEVELOPMENT GUIDELINES AND BEST PRACTICES .................... 5
  Broad-based actionable knowledge from the CWE database ..................... 5
  CERT C guidelines for embedded developers ......................................... 6
  Measuring source code compliance ....................................................... 6

ESTABLISHING A SECURITY-ENHANCED DEVELOPMENT PROCESS .......... 7
  Start with requirements ......................................................................... 7
  Develop based on standards .................................................................. 7
  Test early and test often ....................................................................... 7

EMPLOYING STATIC ANALYSIS ................................................................ 8
  CWE compatible tools ........................................................................ 8
  Putting CWE into practice .................................................................... 9
  CERT C rule checking in practice ......................................................... 9
  Example: Checking code and identifying rule violations ......................... 11

CONCLUSION .......................................................................................... 13
SECURITY AND THE NEED FOR STANDARDS AND BEST PRACTICES

Every day we read about cyber attacks perpetrated against individuals, corporations, or governments. The reasons are varied - competitive, political, military, or economic - but the headlines keep coming. And from attacks like Stuxnet and Duqu, as well as high profile automotive hacks, we’ve come to understand that embedded systems are not immune. In fact, attacks against embedded systems could lead to wide-scale damage to critical infrastructure, including power generation, oil and gas refining, telecommunications, transportation, and water and waste control systems.

While there has been no shortage of investment in cyber security technologies and many advances have been made, much of the effort has been focused on adding security to the perimeter and improving detection technologies and efforts. There is now a growing recognition that software itself needs to be more secure and that security must be built into the fabric of all software. According to a report by the US Department of Homeland Security, “Software Assurance has become critical because dramatic increases in business and mission risks are now known to be attributable to exploitable software.” Dangers increase as software is reused and interfaced with other applications in new environments, which can introduce unintended consequences and increase the number of vulnerable targets. As a result risk exposure is rapidly growing and poorly understood.

Addressing this issue doesn’t mean that we need to be developing and applying new security technologies or that existing safeguards such as encryption are ineffective. It means that the software itself needs to close exploitable gaps. Putting stronger locks on your front door is of no use if the windows are left open. We need to improve the software development process to improve security.

SECURITY IS A PROCESS, NOT A FEATURE

When security is considered a requirement, it is most often addressed by incorporating security features such as encryption and password protection. Otherwise, most software development doesn’t place any significant emphasis on application security. Most organizations provide few if any techniques and tools to help developers produce software that is inherently free of vulnerabilities.

A complete software security perspective not only incorporates protective, post-implementation techniques, but also addresses the need to specify, design, and implement the application so that its attack surface (i.e., the extent of exposure of its vulnerabilities) is minimized. Best practices indicate that a disciplined, repeatable, and security-focused development process should be instituted so that application security measures are incorporated into the design and coding process.
This paper describes how to leverage security databases, secure coding standards, and static analysis tools to enable secure software development and address software security at the source code level.

THE SOFTWARE SECURITY PROBLEM: WHY IT’S SO DIFFICULT TO PRODUCE SECURE SOFTWARE

SECURITY IS JUST NOT A PRIORITY

Most development organizations, perhaps unknowingly, subscribe to the iron triangle adage – “get to market fast, with all the features planned, and a high level of quality…pick two.” And while quality has been part of the conversation, security is typically omitted. The stark reality is that until security becomes a priority, it will receive less attention than the two most visible items on developers’ checklists: features and deadlines. Security has not often been considered a feature or a requirement until recently and as a result it was rarely addressed. Further, when security considerations threaten a deadline, they tend to be avoided.

QUALITY ASSURANCE DOESN’T EQUAL SECURITY ASSURANCE

Traditional quality assurance efforts have not taken security into account. While high quality can reduce security flaws attributable to defects, most traditional software quality assurance does not address intentional malicious behavior. Ensuring that software is secure means ensuring that it can’t be intentionally subverted or forced to fail. It is, in short, software that remains dependable (i.e., correct and predictable) in spite of intentional efforts to compromise that dependability. Developing and testing such software requires the application of proven tools and techniques, which many organizations have not done to date.

EMBEDDED SOFTWARE HAS TOO MANY MOVING PARTS

Embedded systems applications are becoming extremely large and complex, incorporating new and legacy code as well as connectivity components, while running on a variety of operating systems. Adding to the complexity is the fact that software development typically involves multiple geographically distributed development teams, with some or all of the development outsourced to third parties. At the same time, these software-intensive systems are usually resource constrained in terms of power and memory and operate in a more physically harsh environment. Taking into account the software requirements and development process it is difficult enough to ensure that embedded software functions properly, and can be considerably more difficult to ensure that it is secure.

TRAINING IS INADEQUATE

Researchers at Carnegie Mellon University’s Software Engineering Institute have determined that a large percentage of security weaknesses in software could be avoided if developers consciously thought about avoiding them. Unfortunately, many people involved in software development are not taught how to recognize the security implications of certain software requirements or omission of requirements. Nor do many learn the security implications of how software is modeled, architected, designed, implemented, tested, and prepared for distribution and deployment. Without this knowledge, developers may not design and imple-
ment software in line with its specified security requirements, and those requirements may be inadequate in the first place. Further, this lack of knowledge will prevent developers from recognizing and understanding how the mistakes they make during development can manifest as exploitable weaknesses and vulnerabilities in the software when it becomes operational.

NOBODY HAS OWNERSHIP OF SECURITY

Enterprise software applications and networks typically face scrutiny from a variety of players within the IT organization whose jobs include security responsibilities. These individuals may be explicitly tasked with ensuring security, with roles such as chief security officer, enterprise architect, CIO, network administrator, and so on. This focus, however, is largely absent in the embedded world. Relatively few organizations developing embedded software applications have a role explicitly tasked with software security; rather they rely on a variety of roles ranging from product management to engineering to development to QA to get it right.

SECURE DEVELOPMENT GUIDELINES AND BEST PRACTICES

Software attacks have become so frequent and so sophisticated that organizations are overwhelmed by the challenge of figuring out which threats and vulnerabilities pose the greatest risk and how to protect against and respond to those risks.

So what do we know about improving software security? There has been a considerable amount of both commercial and academic research into how to best address cyber security, and the volume of research can be overwhelming to digest. While there are many security task forces, regulations, and standards bodies, among the most prominent security initiatives related to software development are the Common Weakness Enumeration (CWE) database project (http://cve.mitre.org/cwe/) and the CERT C coding standard. The CWE database includes security issues related to multiple programming languages and can be applied to a broad range of application types, including web, desktop, mobile and embedded. In contrast, the CERT C coding standard is focused specifically on the C language, which is the language most widely used in embedded software application development.

BROAD-BASED ACTIONABLE KNOWLEDGE FROM THE CWE DATABASE

According to research done by the National Institute of Standards and Technology (NIST), 64% of software vulnerabilities stem from programming errors. To help identify core weaknesses contributing to software vulnerabilities, the CWE list of common software weaknesses was created as part of a software assurance strategic initiative cosponsored by the National Cyber Security Division of the U.S. Department of Homeland Security. The CWE database is a project that incorporates an international, community-developed formal list of common software weaknesses developed from an all-inclusive database of exploits and vulnerabilities captured over a number of years. CWE builds and maintains a formal list and a classification scheme for software weaknesses. This project is run by the MITRE Corporation and funded by the U.S. Department of Homeland Security.
The CWE compiles an annual list, which prioritizes weaknesses. The top 25 entries are prioritized using input from more than two-dozen different organizations, which evaluate each weakness based on prevalence and importance. The top 25 list also adds a small set of the most effective Monster Mitigations, which help developers to reduce or eliminate entire groups of the top 25 weaknesses, as well as many of the other 800 weaknesses that are documented in the CWE list.

CWE focuses on stopping vulnerabilities at the source by educating designers, programmers, and testers on how to eliminate common mistakes before software is even shipped. The CWE project aims to better understand flaws in software and to create automated tools that can be used to identify, fix, and prevent those flaws.

CERT C GUIDELINES FOR EMBEDDED DEVELOPERS

The CERT C and C++ Secure Coding Standards provide rules and recommendations for secure coding in the C and C++ programming languages. The CERT Division at the Software Engineering Institute (SEI) operated by Carnegie Mellon University publishes these standards. SEI is a research and development center primarily funded by the U.S. Department of Defense and the Department of Homeland Security. During research conducted over 15 years, CERT studied cases of software vulnerabilities and compiled a database. The Secure Coding Initiative uses this database to help develop secure coding practices in C.

CERT provides rulesets for secure coding in C and C++. Each guideline consists of a title, a description, a non-compliant code example, and examples of compliant solutions. The guidelines cover avoiding coding and implementation errors as well as low-level design errors. The aim of these guidelines is to eliminate insecure coding practices and undefined behaviors that can lead to exploitable vulnerabilities. CERT describes a vulnerability as a software defect that affects security when it is present in information systems. The defect may be minor, in that it does not affect the performance or results produced by the software, but nevertheless may be exploited by an attack that results in a significant breach of security. CERT estimates that up to 90% of reported security incidents result from the exploitation of defects in software code or design.

MEASURING SOURCE CODE COMPLIANCE

The CERT Secure Coding Standards can be used as a measure of software security by determining the degree to which a software system complies with the rules and recommendations in this standard. While compliance does not guarantee the absence of vulnerabilities (for example, vulnerabilities resulting from design flaws), it does guarantee the absence of coding errors that are commonly found to be the root causes of vulnerabilities.

The easiest way to validate code as compliant with the CERT Secure Coding standards is to use a certified source-code analysis tool.

The primary aim of CERT C is to enumerate the common errors in C language programming that lead to software defects, security flaws, and software vulnerabilities. The standard provides recommendations about how to produce secure code. Although the CERT guidelines share traits with other coding standards, such as identifying non-portable coding practices, the primary objective is to eliminate vulnerabilities.
ESTABLISHING A SECURITY-ENHANCED DEVELOPMENT PROCESS

START WITH REQUIREMENTS

In the absence of a clear-cut owner for security and a lack of security training and understanding, it becomes imperative to incorporate security into the software development lifecycle (SDLC). A security-enhanced development process takes into account that software will be subjected to hostile conditions and malicious inputs and aims to produce software that will continue to operate dependably under these conditions. Addressing and eliminating exploitable weaknesses starts with the specification of software requirements. From the outset, development teams need to incorporate well-known security best practices such as constraints on process behaviors and handling of input, resistance to and tolerance of intentional failures, and secure multicore designs to prevent unexpected interactions between threads and processes.

DEVELOP BASED ON STANDARDS

Development organizations can make up for a lack of security training by incorporating security standards compliance into their development process. Standards provide developers an accepted approach to coding with a systematic way to identify and avoid risks. By implementing programming standards, organizations take steps to ensure that the code produced across the organization by both internal and outsourced developers is consistent, that the code written by any one developer can be more easily read and understood by any other developer involved in the project, and that code reviews and downstream maintenance will more efficient. This approach provides objective measures and guidelines to follow throughout the defect identification and isolation process, helping diminish latent defects and vulnerabilities and ultimately leading to enhanced code quality. Programming standards help organizations avoid areas of significant programming risk over time, while decreasing rework and lowering overall software development costs.

TEST EARLY AND TEST OFTEN

It is widely documented and acknowledged that finding bugs and defects early in the development process reduces costs and speeds development. The key to early detection is to perform static analysis consistently from the outset of the project on all new code and on any code being reused from a prior project. This approach will find defects when the cost of remediation is at its lowest. Some organizations view the use of static analysis as another hurdle that potentially slows a rapid dash to completing development, but this view is shortsighted. The gains in efficiency from identifying defects early far outweigh the relatively minor extra effort it takes to perform static analysis and act upon the results.

As developers use static analysis tools that include checkers for CWE and documentation that supports the programming standards such as CERT, their knowledge of secure programming practices builds. Using these tools can lead to better recognition of security related issues and bring programming in line with best practices. Static analysis tools that also support role-based and distributed reporting enable teams to prioritize and assign defect remediation tasks, further increasing the pace of development.
EMPLOYING STATIC ANALYSIS

While there are a number of ways to test the security of a software program as it is executed, including dynamic analysis and penetration testing, these methods require running the application and thus target later phases of the development cycle. Static analysis, in contrast, can be performed before the code is even compiled. Code review with a static analysis tool is a vital software security best practice. Because while not all software projects produce specifications or properly documented requirements, they all produce code. Studies have shown that simple implementation errors in code are responsible for up to half of all software security problems, so finding and fixing security-critical bugs at the code level is essential.¹

For developers who lack security training, identifying security problems during a code review can be difficult, if not impossible. Security mistakes can be subtle and easy to overlook even for trained developers. From these two observations, it is easy to see how static analysis tools play a significant role in improving security. A static analysis tool can make the code review process faster and more effective by uncovering security related weaknesses and narrowing the set of potential problems for consideration during a code review.

Using static analysis, developers can identify errors such as memory leaks, access violations, arithmetic errors, and array and string overruns at an early stage to maximize code quality and minimize the impact of errors on the finished product and on the project timeline.

Similarly, while compliance with the CERT C guidelines and identifying CWE coding errors can, in theory, be demonstrated via manual checks, such checks are often impractical for large, complex systems. Automated static analysis tools that can be configured for CWE and SEI CERT coding standards can automate the enforcement of standards based coding and can be deployed with any development methodology from waterfall to agile to hybrid.

CWE COMPATIBLE TOOLS

Static analysis tools that support CWE checks can identify common programming errors that contribute to exploitable vulnerabilities. Support for CWE varies among available tools. More effective tools meet the following criteria:

- **CWE search** - Users can search security elements using CWE identifiers
- **CWE output** - Security elements presented to users include, or enables users to obtain, associated CWE identifiers
- **Mapping accuracy** - Security elements accurately link to the appropriate CWE identifiers
- **CWE documentation** - Documentation describes CWE, CWE compatibility, and how CWE-related functionality is used

Products that meet these criteria can be reviewed and registered as officially "CWE-Compatible."²

² https://cwe.mitre.org/compatible/
PUTTING CWE INTO PRACTICE

CWE compatible static analysis tools report on all known types of weaknesses. They are used by developers and managers during code reviews to gain insight into code quality and security as well as to make corrections as required. The following table shows checks that a properly configured static analyzer can perform:

<table>
<thead>
<tr>
<th>A-STATIC ANALYSIS CHECKS</th>
<th>EXAMPLE WEAKNESS FOUND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boundary checks</td>
<td>static/dynamic buffer overflows/underflows (CWE-125, CWE-120, CWE-170, CWE-124) types incompatibilities (signed/unsigned) (CWE-194)</td>
</tr>
<tr>
<td>Resource leak checks</td>
<td>memory leaks (CWE-404)</td>
</tr>
<tr>
<td></td>
<td>descriptor leaks (CWE-404)</td>
</tr>
<tr>
<td>Memory safety checks</td>
<td>dereference null (CWE-476), use after free (CWE-416)</td>
</tr>
<tr>
<td></td>
<td>double free (CWE-415), bad free (CWE-590)</td>
</tr>
<tr>
<td>Dead code checks</td>
<td>unnecessary code (CWE-561)</td>
</tr>
<tr>
<td></td>
<td>wrong error check (CWE-252, CWE-665, CWE-569)</td>
</tr>
<tr>
<td>Uninitialized/unused variables checks</td>
<td>unnecessary variable handling (CWE-563)</td>
</tr>
<tr>
<td></td>
<td>using uninitialized variable (CWE-457, CWE-456)</td>
</tr>
<tr>
<td>Race conditions / synchronization checks</td>
<td>TOCTOU (time of check / time of use) (CWE-367)</td>
</tr>
<tr>
<td></td>
<td>unsynchronized access to shared data in multithread apps (CWE-362)</td>
</tr>
<tr>
<td></td>
<td>issues with locks (CWE-362)</td>
</tr>
<tr>
<td>Human coding errors</td>
<td>various cut&amp;paste issues, missing breaks (CWE-484)</td>
</tr>
<tr>
<td></td>
<td>priority of operators issues (CWE-569)</td>
</tr>
<tr>
<td></td>
<td>stray semicolon after if (CWE-398)</td>
</tr>
<tr>
<td></td>
<td>missing asterisks in pointer operations (CWE-476)</td>
</tr>
</tbody>
</table>

CERT C RULE CHECKING IN PRACTICE

Automated static analysis tools are a cost-effective and efficient way to assess compliance with CERT C guidelines. With the ability to track and document compliance to this standard, managers can chart their progress towards secure code development internally and provide object evidence of this progress externally.
A technical report from SEI explains:

The CERT secure coding standard consists of rules and recommendations. Coding practices are defined to be rules when all of the following conditions are met:
1. Violation of the coding practice will result in a security flaw that may result in an exploitable vulnerability.
2. There is a denumerable set of conditions for which violating the coding practice is necessary to ensure correct behavior.
3. Conformance to the coding practice can be determined through automated analysis, formal methods, or manual inspection techniques.

Rules must be followed to claim compliance with a standard unless an exceptional condition exists. If an exceptional condition is claimed, the exception must correspond to a predefined exceptional condition, and the application of this exception must be documented in the source code.

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.

Compliance with recommendations is not necessary to claim compliance with a coding standard. It is possible, however, to claim compliance with one or more verifiable guidelines. 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.

The CERT standards group rules as follows:

<table>
<thead>
<tr>
<th>RECOMMENDATIONS</th>
<th>RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 01. Preprocessor (PRE)</td>
<td>Rule 10. Environment (ENV)</td>
</tr>
<tr>
<td>Rule 02. Declarations and Initialization (DCL)</td>
<td>Rule 11. Signals (SIG)</td>
</tr>
<tr>
<td>Rule 03. Expressions (EXP)</td>
<td>Rule 12. Error Handling (ERR)</td>
</tr>
<tr>
<td>Rule 05. Floating Point (FLP)</td>
<td>Rule 14. Concurrency (CON)</td>
</tr>
<tr>
<td>Rule 06. Arrays (ARR)</td>
<td>Rule 48. Miscellaneous (MSC)</td>
</tr>
<tr>
<td>Rule 07. Characters and Strings (STR)</td>
<td>Rule 50. POSIX (POS)</td>
</tr>
<tr>
<td>Rule 08. Memory Management (MEM)</td>
<td>Rule 51. Microsoft Windows (WIN)</td>
</tr>
<tr>
<td>Rule 09. Input Output (FIO)</td>
<td></td>
</tr>
</tbody>
</table>

3 repository.cmu.edu/cgi/viewcontent.cgi?article=1301&context=sei
EXAMPLE: CHECKING CODE AND IDENTIFYING RULE VIOLATIONS

Developers can use static analysis tools to identify potential CERT rule violations in their source code. For example, consider the Category 4 rule, INT32-C: Ensure that operations on signed integers do not result in overflow. CERT documents this rule as follows:

Signed integer overflow is undefined behavior. Consequently, implementations have considerable latitude in how they deal with signed integer overflow (see MSC15-C, Do not depend on undefined behavior). An implementation that defines signed integer types as being modulo, for example, need not detect integer overflow. Implementations may also trap on signed arithmetic overflows, or simply assume that overflows will never happen and generate object code accordingly. It is also possible for the same conforming implementation to emit code that exhibits different behavior in different contexts. For example, an implementation may determine that a signed integer loop control variable declared in a local scope cannot overflow and may emit efficient code on the basis of that determination, while the same implementation may determine that a global variable used in a similar context will wrap.

For these reasons, it is important to ensure that operations on signed integers do not result in overflow. Of particular importance are operations on signed integer values that originate from a tainted source and are used as

1. Integer operands of any pointer arithmetic, including array indexing;
2. The assignment expression for the declaration of a variable length array;
3. The postfix expression preceding square brackets [] or the expression in square brackets [] of a subscripted designation of an element of an array object; and
4. Function arguments of type size_t or rsize_t (for example, an argument to a memory allocation function).

Integer operations will overflow if the resulting value cannot be represented by the underlying representation of the integer. The following table indicates which operations can result in overflow.

Static analysis can identify instances of signed integer overflow rule violations and classify each instance as definite, apparent, or suspicious.

A definite rule violation will happen every time the code is executed. For example, the highlighted line of code in Figure 1 will always result in a signed integer overflow.
Figure 1. Static analyzer output for a definite signed integer overflow.

An *apparent* rule violation will happen unless a specific path elsewhere in the code, associated with a conditional operation, is always bypassed. For example, the highlighted line of code in Figure 2 is an apparent signed integer overflow.

Figure 2. Static analyzer output for an apparent signed integer overflow.
A suspicious rule violation will happen if certain conditions are not fulfilled in the execution of a loop construct. For example, the highlighted line of code in Figure 3 is a suspicious signed integer overflow.

![Figure 3. Static analyzer output for a suspicious signed integer overflow.](image)

**CONCLUSION**

Improving and ensuring the security of embedded software applications requires more than just adding security features. It depends heavily on the adoption of tools and techniques to eliminate vulnerabilities introduced during programming. Organizations can now automate the enforcement of well-documented coding standards to enable more secure software development. Secure coding standards provide an objective and uniform set of rules and guidelines based on years of security research and established best practices. Following these standards and the guidelines they provide minimizes insecure coding practices and undefined behaviors that can lead to exploitable software vulnerabilities.

With today’s increasingly large and complex code bases, software developers cannot rely on manual inspection and code review alone. Automated static analysis enables organizations to cut costs and speed development by enforcing coding standards, streamlining code reviews, and improving overall system security before a product is released.
ABOUT PRQA

Established in 1985, PRQA is recognized throughout the industry as a pioneer in static analysis, championing automated coding standard inspection and defect detection, delivering its expertise through industry-leading software inspection and standards enforcement technology.

PRQA static analysis tools, QA·C and QA·C++, are at the forefront in delivering MISRA C and MISRA C++ compliance checking as well as a host of other valuable analysis capabilities. All contain powerful, proprietary parsing engines combined with deep accurate dataflow that deliver high fidelity language analysis and comprehension. They identify problems caused by language usage that is dangerous, overly complex, non-portable or difficult to maintain. Additionally, they provide a mechanism for coding standard enforcement.

CONTACT US

PRQA has offices globally and offers worldwide customer support. Visit our website to find details of your local representative.
Email: info@programmingresearch.com
Web: www.programmingresearch.com

All products or brand names are trademarks or registered trademarks of their respective holders.