A Guide through the Jungle of Security Coding Standards

Software Security is Becoming Increasingly Important

Michal Rozenau, Parasoft

Searching for security-focused coding standards, you find a variety of sources – including CERT Coding Standards, OWASP, CWE, and multiple various recommendations and best practices. Additionally, there are many domain-specific standards, including MISRA, AUTOSAR, and a whole family of IEC 61508-based standards. It is a challenge to deal with this amount of information and determine the set of coding standards that should be applied to your specific project. And it is an even bigger challenge to do it in the middle of the SDLC, when the already-existing software needs to be suddenly tuned to comply to such standard.

Background

Software plays a more and more important role in our everyday lives. Whether it is a plane, a car, a medical device, a mobile phone, a refrigerator or a watch – it has some software in it. Whether it is a bank or insurance company, an energy plant or a traffic control system – software is an essential part of it. The existence of this software gives us many opportunities – we can live in smart homes, driving smart cars, and our smartphones and smartwatches assist us with everything we need. Additionally, more and more of these elements are getting connected to each other, which may cause additional benefits and make our life much easier.

But all of that comes with a risk. Software security holes can be exploited to gain unprivileged access to any system. This may mean simple hacks like turning off our lights when we don't expect it, spying on us with our cameras, or even emptying our bank accounts without our knowledge. Software bugs, which may result in undesired behavior of the application, also may cause similar issues, even without participation of the cybercriminals. Therefore, software security is becoming increasingly important.

From CVE to CWE Top 25

In 1999, MITRE Corporation (an American not-for-profit organization that operates research and development centers sponsored by the federal government) started to document known software security vulnerabilities on the Common Vulnerabilities and Exposures (CVE) list. The initial list consisted of 321 CVE entries, but it grew quickly, and currently (as of September 2018), the CVE list contains over 100,000 entries and is maintained by 92 CVE Numbering Authorities (CNAs) – different organizations from all around the world that are authorized to assign CVE IDs to detected issues. These organizations include vulnerability researchers, product vendors, as well as bug bounty programs. The CVE list is very widely used and is recommended by the National Institute of Standards and Technology (NIST), the U.S. Defense Information Systems Agency (DISA), and many more.

As the CVE list was growing, it became necessary to categorize its entries into groups depending on the type of the issues to better understand the roots of the most common problems and to take appropriate actions to avoid similar issues in the future. To do this, MITRE Corporation started the work on the categorization of the known issues, which ended with the publication of the Preliminary List of Vulnerability Examples for Researchers (PLOVER) document. Combining this work with other researches resulted in defining Common Weakness Enumeration (CWE) list which is a formal list of software weakness types.

Currently, CWE List Version 3.1 contains around 700 types of weaknesses grouped into 250 categories and views. Since this number can be overwhelming, the “Top 25 Most Dangerous Software Errors” list is maintained by the CWE in conjunction with the SANS Institute. That list collects the most widespread and critical errors that can lead to serious vulnerabilities in software.

The top of the current “Top 25” list consists of:

1. CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
2. CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
3. CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
4. CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
5. CWE-306: Missing Authentication for Critical Function

Functional Safety Standards

There are industries in which safety is much more important than in others. More safety is expected from the airborne, automotive, or healthcare systems than from the television or other entertainment systems. To ensure the safety of such computer systems, International Electrotechnical Commission (IEC) developed the IEC 61508: Functional Safety of Electrical / Electronic / Programmable Electronic Safety-related Systems as a standard applicable across all industries. It is not specific to software only but considers all aspects of a computer system. IEC 61508 Part 3, Software requirements, is specifically focused on the software part of the system and contains multiple requirements for all phases of the software development lifecycle. IEC 61508-3, in Table A.9 – Software verification, describes list of the recommended techniques to be used to perform software verification and explicitly states that static analysis is highly recommended for the higher SILs.

Further, in Table B.1 – Design and coding standards, IEC 61508-3 presents the list of specific software design techniques recommended (R) and highly recommended (HR) for particular SILs:

Technique/Measure	SIL 1	SIL 2	SIL 3	SIL 4
Use of coding standard to reduce likelihood of errors	HR	HR	HR	HR
No dynamic objects	R	HR	HR	HR
No dynamic variables	R	HR	HR	
Online checking of the installation of dynamic variables	--	R	HR	HR
Limited use of interrupts	R	R	HR	HR
Limited use of pointers	--	R	HR	HR
Limited use of recursion	--	R	HR	HR
No unstructured control flow in programs in higher level languages	R	HR	HR	HR
No automatic type conversion	R	HR	HR	HR

IEC 61508 is a general-purpose standard used in the variety of industries. There are also standards dedicated for the specific industries, e.g.:

- Airborne: DO-178C / ED-12C
- Automotive: ISO 26262
- Railway: IEC 62279 / EN 50128
- Nuclear Power Plants: IEC 61513

These standards consider specifics of a given domain, but their general idea – at least from the software point of view – is quite similar. All of them require – among other verification techniques – static analysis to be performed on the developed code and they provide some high-level guidelines on what needs to be done, but on the other hand leave lots of room for interpretation. They usually do not name any specific coding conventions or coding standards to be used, but e.g. ISO 26262 mentions MISRA C as an example of a coding guideline for the C programming language.

(Secure) Coding Standards

Writing secure code means writing code that will not be vulnerable, i.e. code that does not contain weaknesses that could potentially be exploited. It means writing code that complies with some “safe” patterns and the code that avoids “unsafe” patterns. And these patterns will be different for different programming languages. Of course, there is no single golden set of rules that could be followed to guarantee safety of the software. Some of the widely used coding standards that consider safety are:

- For C language: MISRA C, SEI CERT C Coding Standard
- For C++ language: MISRA C++, JSF AV C++ Coding Standard, SEI CERT C++ Coding Standard, AUTOSAR C++ Coding Guidelines

MISRA C and MISRA C++ standards are developed by the MISRA (Motor Industry Software Reliability Association). AUTOSAR C++ Coding Guidelines are developed by the AUTOSAR (AUTomotive Open System ARchitecture) development partnership.

JSF AV C++ Coding Standard, developed by the Lockheed Martin Corporation. SEI CERT C and C++ Coding Standards contain general rules meant to ensure the safety, reliability, and security of software systems developed in the C and C++ programming languages.

Rules in the standards are usually grouped into multiple categories to allow easier navigation, as the number of rules in given standard may be quite large, e.g.:

Standard	# of guidelines	Details
MISRA C 2012	173	156 rules, 17 directives
CERT C	307	121 rules, 186 recommendations
AUTOSAR	344	319 required, 25 advisory
CERT C++	163	83 C++ rules, 80 relevant C rules

Best practices

Considering the number of different functional safety standards, coding standards, and number of guidelines recommended or required by each standard, it is important to make good choices when starting the initiative of making the code secure.

If the software needs to be certified according to a specific functional safety standard (e.g. ISO 26262 for automotive or DO-178C for airborne), the initial decision is already made. But regardless, a choice needs to be made to determine the coding standard, or standards, or a subset of a standard to be enforced on the developed software. It may, but does not have to be, one of the standards mentioned above.

Then, an appropriate static analysis tool needs to be chosen, as it is not possible to verify compliance with all these rules manually. There are both open source and commercial static analysis tools available. It is important to pick up a good one. Some of the factors that should be considered are:

- Does the tool support the chosen coding standard(s) – fully / partially?
- If the tool qualification is required by the functional safety standard, is the tool certified? Does it provide the qualification kit?
- Can the tool produce analysis reports in a form required to do compliance analysis?
- Can the tool produce analysis reports in a form easy to read by the developers?
- Does the tool integrate cleanly with used IDEs, build and CI systems?
- Does the tool allow for the selective analysis of the new code, changed code or the legacy code?
- Does the tool support flexible configuration of the analysis?
- Does the tool take advantage of the risk scoring algorithms to help prioritize found defects?

Additionally, the CWE Community drives CWE Compatibility and Effectiveness Program, which is a formal review and evaluation process for a product or a service. The list of "Officially CWE-Compatible" products and services contains currently 55 entries. Using the tool from this list ensures it has achieved the final stage of MITRE's formal CWE Compatibility Program.

When the coding standard and appropriate tool are chosen, for the software projects started from scratch further work should be easy – simply integrate the tool with the CI system and make sure that no defect is left unattended.

But usually, coding standard needs to be enforced on the software already under development. In such cases blind execution of the analysis on the whole code base may produce hundreds of defects being reported, which are hard to handle all together. Fortunately, there are multiple techniques that can be used to ease the process.

It is recommended to focus on the newly written or modified code first, to make sure that at least no new defects are introduced from the moment coding standard has been established.

Another good practice is to prioritize the reported defects to make sure the most severe ones are handled in the first place. As an example, CERT C and C++ Rules have risk assessment associated with them allowing for easy prioritization of found defects, e.g.:

Rule	**Severity**	**Likelihood**	**Remediation Cost**	**Priority**	**Level**
FIO30-C	High	Likely	Medium	**P18**	**L1**
FIO32-C	Medium	Unlikely	Medium	**P4**	**L3**
FIO34-C	High	Probable	Medium	**P12**	**L1**
FIO37-C	High	Probable	Medium	**P12**	**L1**
FIO38-C	Low	Probable	Medium	**P4**	**L3**

Prioritized list of defects together with the configurable list of rules to be enforced help focusing on the rules with the highest severity first, increasing the number of rules as the more severe defects are getting fixed.

The most important part is to make sure that analysis is followed by appropriate actions taken by the development team. It does not matter how good the reports from the static analysis are if they are not used by the developers to fix the code, leading to more safe and secure software products.

Bibliography

[1] Hicken Arthur, Parasoft, *Embedded Cybersecurity Through Secure Coding Standards CWE and CERT*, Published on: https://alm.parasoft.com/embedded-cybersecurity-through-secure-coding-standards-cwe-and-cert

[2] MITRE Corporation, *Common Vulnerabilities and Exposures (CVE)*, Published on https://cve.mitre.org/

[3] MITRE Corporation, *Common Weakness Enumeration (CWE)*, Published on https://cwe.mitre.org/

[4] IEC, *IEC 61508-3:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 3: Software requirements*

[5] ISO, *ISO 26262-6:2011(en) Road vehicles - Functional safety - Part 6: Product development at the software level*

[6] Carnegie Mellon University, *SEI CERT C Coding Standard*, Published on https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard

[7] Carnegie Mellon University, *SEI CERT C++ Coding Standard*, Published on https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682

Author

Michal Rozenau is a Project Lead Engineer at Parasoft. He finished his MSc studies in Computer Science at the AGH University of Science and Technology in Krakow in 2003. Since then, he gained software development experience using C, C++, Java and C# languages. He also specialized in applying Parasoft's products to use in the safety related applications complying to safety standards such as IEC 61508, ISO 26262, DO-178B/C, EN 50128.