WIND™

AN INTEL COMPANY

The CWE/SANS Top 25 Security Vulnerabilities: What They Mean for Embedded Developers

WHEN IT MATTERS, IT RUNS ON WIND RIVER

EXECUTIVE SUMMARY

The Common Weakness Enumeration (CWE/SANS) Top 25 "Most Dangerous Software Errors" list is a well-known compilation of the most common security vulnerabilities found across all types of systems. However, embedded developers might dismiss the importance of the list because it includes many types of vulnerabilities that aren't applicable to their domain. This paper highlights the key vulnerabilities that embedded developers need to be aware of and how they can affect the security of their device. The role of automated tools and how they can mitigate risk through early detection and analysis, consistent and automated testing, and more efficient software development is also covered. The real financial benefit of early detection and analysis is discussed as a key motivator for improving tool usage and development practices.

TABLE OF CONTENTS

INTRODUCTION

The CWE/SANS Top 25 is fairly well known among security experts, but might be overlooked by embedded developers since the list covers all types of systems and programming languages. Developers are fully aware of the quality impacts of many of these errors, but they may know less about the security implications. The classic example is the buffer overflow error: All programmers know this error can cause a program to crash or become unresponsive, but many developers fail to realize that an attacker can trigger these errors to execute code, reveal data, or cause a denial of service.

The Top 25 list is a "who's who" of dangerous errors that are the most commonly reported security vulnerabilities. Knowledge is power, and knowing how to mitigate the risk of these errors is the first step to improving your device's security. To reduce risk, adopting an "earlier is better (and cheaper)" approach is prudent. After discussing some of the top software security vulnerabilities, this paper discusses the use of a security improvement framework that can greatly reduce the time and effort required to find, analyze, and fix these bugs as early in the development lifecycle as possible.

THE CWE/SANS TOP 25

The CWE/SANS Top 25 most dangerous software errors are listed below. The errors marked with an asterisk are applicable to embedded systems (but also apply to networked, dedicated, and consumer devices). It may be surprising to the embedded developer to discover that a majority of these errors do in fact apply to the types of systems they develop, or that these types of errors are exploitable from a security perspective.

Table 1. CWE/SANS Top 25 Most Dangerous Software Errors

Rank	Table Head	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements Used in an SQL Command ('SQL Injection')
[2]*	83.3	CWE-78	Improper Neutralization of Special Elements Used in an OS Command ('OS Command Injection')
[3]*	79.0	CWE-120	Buffer Copy Without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-Site Scripting')
[5]*	76.9	CWE-306	Missing Authentication for Critical Function
[6]*	76.8	CWE-862	Missing Authorization
[7]*	75.0	CWE-798	Use of Hard-Coded Credentials
[8]*	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]*	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]*	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]*	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]*	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]*	68.5	CWE-494	Download of Code Without Integrity Check
[15]*	67.8	CWE-863	Incorrect Authorization
[16]*	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere
[17]*	65.5	CWE-732	Incorrect Permission Assignment for Critical Resource
[18]*	64.6	CWE-676	Use of Potentially Dangerous Function
[19]*	64.1	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
[20]*	62.4	CWE-131	Incorrect Calculation of Buffer Size
[21]*	61.5	CWE-307	Improper Restriction of Excessive Authentication Attempts
[22]	61.1	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
[23]*	61.0	CWE-134	Uncontrolled Format String
[24]*	60.3	CWE-190	Integer Overflow or Wraparound
[25]*	59.9	CWE-759	Use of a One-Way Hash Without a Salt

THE BIG 10 CODING ERRORS AND THEIR IMPACT FOR EMBEDDED DEVELOPERS

Some of the Top 25 are more applicable to embedded developers than others and therefore have a higher priority for them. Also, some of the errors are due to coding, while others are due to configuration (or a combination of both). Wind River® has identified the vulnerabilities that are most likely to impact embedded developers from this list. The following list describes the "Big 10" errors that embedded developers are likely to encounter.

CWE-120: Buffer Copy Without Checking Size of Input ('Classic Buffer Overflow')

The buffer overflow is the most common coding error embedded developers are likely to run into, and it can have serious security consequences if exploited. A buffer overflow is a run-time error where the index of a buffer accesses memory outside the bounds of the buffer, array, string, and so on. Embedded developers need to be particularly aware of buffer overflow errors since much of their coding is done in C or assembler language, where such errors are commonplace because there are not any strict bounds-checking mechanisms. Also, embedded developers may not be in the habit of checking input data (even from within the system, from disk or flash storage, for example) to ensure it isn't out of bounds.

CWE-250: Execution with Unnecessary Privileges

A typical security issue is that tasks, threads, or processes running at high privilege levels (e.g., root in Linux) have privileged access to files and devices and can execute any command or system API. If such a process is subject to an attack leveraging one of the many other security vulnerabilities (for example, a malicious code injection), the injected malware runs at the highest system privilege level. Only essential code should be running at the high privilege level (or kernel level in an RTOS)—vulnerabilities at this level would do the most damage.

CWE-311: Missing Encryption of Sensitive Data

Critical data in your system should never be stored or transmitted in clear text. Any data stored in files, stored on flash memory, or transmitted over the network should be encrypted. Private user information, system control, and sensitive data in particular need to be encrypted. In the past, embedded systems have relied on their isolation from a public network and relative obscurity to keep data safe. This is no longer acceptable; any device that can potentially store sensitive information should use encryption. For example, home medical devices should store all patient data in encrypted form. Any transmission of this data over a mobile or home network should also be encrypted.

CWE-807: Reliance on Untrusted Inputs in a Security Decision

Any input from outside the system must be considered untrusted. Using any such data as input into critical functions can have serious security ramifications. If this input is used in security decisions within the code, it can be exploited. Although embedded systems may not have human input directly, any data from files, networks, or peripheral devices is untrusted and needs to be validated before use.

CWE-306: Missing Authentication for Critical Function

Authentication should be required for critical functions in the system. Creating or updating sensitive information should be done on behalf of an authenticated user, for example. Also, any function that has critical system impact or could possibly consume a large amount of system resources should be authenticated. Embedded systems may not have authentication in place per se, but if critical functions are being performed on behalf of user input (e.g., from an HMI) then that user should be authenticated.

CWE-494: Download of Code Without Integrity Check

Embedded systems are likely to need patching and upgrades out in the field, but applying patches should only be done with validated code from the manufacturer. Without proper integrity checks of upgrades or downloaded applications, it's possible to insert malware into the system. To combat this danger, downloaded applications and code could be encrypted and signed by the manufacturer.

CWE-798: Use of Hard-Coded Credentials

Relying on hard-coded credentials to access a system's user interface or network-based shell is, unfortunately, a commonplace problem in embedded systems—a method often used by manufacturers for back-door access to a device. Unless a manufacturer or system integrator changes the defaults (or forces initialization), these hard-coded credentials end up in the field. Manufacturers must configure embedded systems to avoid hard-coded credentials, especially when devices are deployed.

CWE-327: Use of a Broken or Risky Cryptographic Algorithm

There are factors inhibiting modern encryption in embedded systems: processing power to encrypt/decrypt data, especially in real time; OS support for up-to-date cryptographic algorithms; and long product life spans where outdated techniques are still used. Cryptographic libraries in embedded systems need to be kept up to date as technology changes, and devices need to be configured to use the strongest yet most computationally efficient encryption possible. At the same time, embedded system processing power has increased significantly, even in low-power devices, making the overhead of cryptography less onerous.

CWE-30: Improper Restriction of Excessive Authentication Attempts

Although not unique to embedded systems, it can be particularly dangerous not to limit the number of authentication attempts (e.g., user login attempts for a controller user interface). A combination of hard-coded credentials and weak encryption makes brute force authentication attacks feasible on embedded devices. Limiting the number of attempts is a simple way to thwart, or at least slow down, brute force attacks.

CWE-22: Path Traversal

If user, file, or network input data is used to construct a path name, it is possible to traverse the device's directory structure if proper input checks are not performed. This vulnerability can allow the leaking of private and sensitive information, the replacement of libraries and executables with malware, or corrupt system settings. As with any external data to the system, it must be validated before use.

THE MONSTER MITIGATIONS

The CWE/SANS Top 25 also lists the so-called Monster Mitigations—the top things you can do to mitigate software security risks in your devices. The Monster Mitigations are listed below.

Table 2. Monster Mitigations

ID	Description
M1	Establish and maintain control over all of your inputs.
M2	Establish and maintain control over all of your outputs.
M3	Lock down your environment.
M4	Assume that external components can be subverted, and your code can be read by anyone.
M5	Use industry-accepted security features instead of inventing your own.

Table 3. Monster Mitigations, cont'd.

ID	Description
GP1	(general) Use libraries and frameworks that make it easier to avoid introducing weaknesses.
GP2	(general) Integrate security into the entire software development lifecycle.
GP3	(general) Use a broad mix of methods to comprehensively find and prevent weaknesses.
GP4	(general) Allow locked-down clients to interact with your software.

The mitigations of special note for software tool automation are M4, M5, GP1, GP2, and GP3. These are enumerated below:

- **M4: Assume that external components can be subverted, and your code can be read by anyone.** Far too often, embedded developers assume their device operates in isolation on private networks, and "security by obscurity" is relied upon to avoid proper system hardening. Embedded developers need to adopt security as a key ingredient in their development lifecycle and assume that even private intellectual property can be reverse engineered and exposed.
- **M5: Use industry-accepted security features instead of inventing your own.** Leveraging standards-based encryption or built-in OS security features is safer and more cost effective than reinventing the wheel. Inventing new security features is error prone and more likely to be exploitable than well-known and well-tested off-the-shelf capabilities.
- **GP1: Use libraries and frameworks that make it easier to avoid introducing weaknesses.** Many classes of bugs are due to using unsafe function calls. In fact, the security issues with these types of functions are well known, yet they still get used. Leveraging safe string libraries, for example, can prevent whole classes of buffer overflow errors because bounds checking is built in.
- **GP2: Integrate security into the entire software development lifecycle.** Security must be built in, not a tacked-on feature added after the fact. Security needs to be considered during the inception phase of every product. Embedded devices no longer function in isolation—even small, low-power devices connect to some kind of network. The demand for interconnection, remote control and updating, and data harvesting has created a much greater focus on security for embedded devices. It is critical that security become a first class citizen in the development lifecycle.

- **GP3: Use a broad mix of methods to comprehensively find and prevent weaknesses.** This recommendation speaks to the use of tools and techniques to automate the detection and analysis of security vulnerabilities. By no means is this the only approach, however—in fact, a holistic approach that incorporates the product development lifecycle, run-time and hardware environment, tools, and people is needed.

OPTIMIZATIONS AND THE 'PERFORMANCE TAX'

Embedded systems are the product of many engineering and business constraints. Often these systems have significant performance, memory, network, and storage constraints that are unusual in IT or enterprise systems, and that limit the ability to achieve security requirements. Furthermore, embedded systems historically had little or no security capabilities. Adding better network support or encryption to existing devices can be impossible due to the hardware constraints. Despite this constraint on legacy systems, however, developers of newly designed systems need to increase the performance envelope for security, as they have for all other requirements. Embedded device manufacturers can't afford to bolt on security after a product has shipped.

CODE VS. CONFIGURATION

Many of the critical security vulnerabilities are configuration issues that are not related to source code. For example, using hard-coded credentials is a common problem in embedded systems, yet it is the choice of the manufacturer to configure the end system this way. It is absolutely critical that system configuration be given a high priority in the secure design of the product.

SOFTWARE DEVELOPMENT TOOL AUTOMATION TO IMPROVE SECURITY

Because security must be built in at the inception of an embedded device, tools and appropriate development processes play a big role as part of a holistic, business-wide approach to reducing the security risk in embedded systems. For more information on this holistic approach, see the Wind River white paper **"Five Steps to Improving Security in Embedded Systems."**

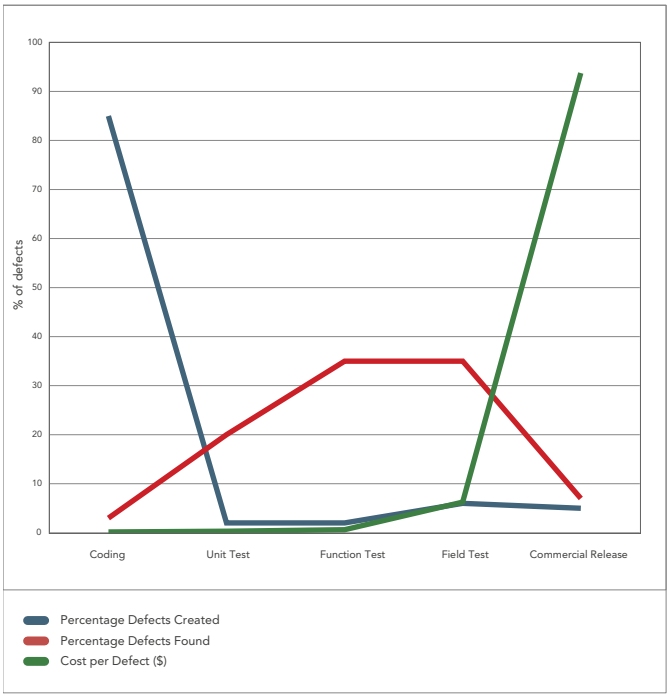

Figure 1. Defects and costs at different stages of the lifecycle

THE PAYOFF FOR AUTOMATION AND FIXING VULNERABILITIES EARLY

The return on investment of finding security vulnerabilities is huge. It has been estimated that a single vulnerability can cost a company as much as $300,000 to fix (Aberdeen Group, "Security and the Software Development Lifecycle: Secure at the Source," 2010). Another analysis showed that public disclosure of a vulnerability can impact a company's stock price by as much as 0.6% per vulnerability (Telang & Wattal, 2007).

Fixing vulnerabilities early has an equally huge cost savings. Consider the relatively short time it takes to add a security requirement to a project. For example, adding the requirement to add a network firewall to the embedded device could prevent many types of network attacks. But adding and enabling the firewall once a device is built is very expensive and prohibitive if the device is already in the field. Or consider the small amount of time it takes

to run a static analysis on code submitted into your source repository—minutes of developer time, versus the weeks or months required to fix the same issues in a manufactured product. The following graph shows the relative cost savings at different phases of the lifecycle (based on many types of projects and industries, most likely using traditional development techniques.)

CONCLUSION

The CWE/SANS Top 25 is an important resource for programmers, including embedded developers. A majority of these security vulnerabilities apply to embedded systems, and Wind River has identified the most significant 10. Mitigation strategies are key to addressing the security risk to your device. An important strategy for manufacturers is to adopt a combination of secure design techniques, automated tools, and staff awareness and training. Finding and fixing security vulnerabilities in the field is very expensive, so it is critical that these errors are found as early as possible. The financial, scheduling, and risk reduction benefit of early detection is extremely high, making the case for tools, process, and training investment. Security must be viewed as an entire device lifecycle issue, and must be built in from the beginning; bolting on security after a product is nearly ready for production is a recipe for disaster.

WIND
AN INTEL COMPANY

Wind River is a global leader in delivering software for the Internet of Things. The company's technology is found in more than 2 billion devices, backed by world-class professional services and customer support. Wind River delivers the software and expertise that enable the innovation and deployment of safe, secure, and reliable intelligent systems.