DEVELOPING SECURE EMBEDDED SOFTWARE

QUALITY DOESN’T EQUAL SECURITY
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td>3</td>
</tr>
<tr>
<td>SOFTWARE IS EVERYWHERE AND TIME TO MARKET IS EVERYTHING</td>
<td>3</td>
</tr>
<tr>
<td>THE SOFTWARE SECURITY PROBLEM:</td>
<td></td>
</tr>
<tr>
<td>COMMON PITFALLS AND MISCONCEPTIONS</td>
<td>4</td>
</tr>
<tr>
<td>1. Misconception 1: Security by Obscurity is a Valid Strategy</td>
<td>4</td>
</tr>
<tr>
<td>2. Misconception 2: Security Features Equal Secure Software</td>
<td>4</td>
</tr>
<tr>
<td>SECURE DEVELOPMENT APPROACHES AND TESTING TOOLS</td>
<td>6</td>
</tr>
<tr>
<td>Identifying and Correcting Security Flaws</td>
<td>6</td>
</tr>
<tr>
<td>Risk and Threat Management Tools</td>
<td>7</td>
</tr>
<tr>
<td>Application Lifecycle Management Tools</td>
<td>7</td>
</tr>
<tr>
<td>System Simulation</td>
<td>8</td>
</tr>
<tr>
<td>Robustness and Fuzz Testing Tools</td>
<td>8</td>
</tr>
<tr>
<td>Dynamic Code Analysis Testing Tools</td>
<td>8</td>
</tr>
<tr>
<td>Static Code Analysis Testing Tools</td>
<td>8</td>
</tr>
<tr>
<td>SUMMARY</td>
<td>9</td>
</tr>
</tbody>
</table>
INTRODUCTION

Many organizations are only now becoming aware of the need to incorporate security into their software development lifecycle. Raising awareness of common pitfalls is the first step to avoid falling prey to them, but awareness by itself is insufficient. Understanding security is one thing; applying that understanding in a complete and consistent fashion to meet security goals is quite another. Effectively addressing embedded software security requires a combination of people, process, and technology. No single tool, technique, or process will ever provide a complete solution. This paper explains why some commonly used approaches to security typically fail and outlines a development strategy for getting security right.

SOFTWARE IS EVERYWHERE AND TIME TO MARKET IS EVERYTHING

We are all increasingly dependent on software, and the security and reliability of that software will influence growth and profitability for manufacturers – putting pressure on development teams to produce better code faster.

The ever-expanding array of cutting-edge devices, including smartphones, tablets, electric cars, new aircraft, and wearable devices, is driving a rapid expansion of the amount of embedded software applications we rely upon in our daily lives. The Internet of Things (IoT) revolution is set to dramatically alter many industries in 2016, with the promise of valuable new revenue streams opened by the software running on embedded devices.

Intelligent device manufacturers are increasingly recognizing the need to become software-centric companies. More and more, the value of physical devices will be defined by the embedded software inside them. Manufacturers will have to start thinking and acting more like software companies, seeing the software applications they build into their products as a driver to reduce manufacturing costs, increase product innovation, and capture new revenue streams. More manufacturers will move away from products based on fixed-function, disconnected devices to those based on flexible, seamlessly connected systems.

IoT and embedded device attacks will continue to make headlines – increasing pressure on device makers to focus on security and remediation. Recently, much attention has been paid to potential security threats facing smart Internet-connected devices – from refrigerators to vehicles. As incidents of these device hacks mount, it has become increasingly apparent that device makers will need to make security a higher priority – both to prevent attacks and to quickly remediate the problem when attacks do occur. Device makers will need to adopt measures to make devices more tamper resistant and to make secure software that is free from exploitable vulnerabilities.

THE INTERNET OF THINGS

Gartner, Inc. forecasts that in 2016, 5.5 million new things will get connected every day, noting "(IoT), which excludes PCs, tablets, and smartphones, will grow to 26 billion units installed in 2020 and that 6.4 billion connected things will be in use worldwide in 2016, up 30 percent from 2015, and will reach 20.8 billion by 2020."
THE SOFTWARE SECURITY PROBLEM: COMMON PITFALLS AND MISCONCEPTIONS

MISCONCEPTION 1: SECURITY BY OBSCURITY IS A VALID STRATEGY

Many embedded engineers have long assumed that embedded devices are not targets for hackers. These assumptions are often predicated on outdated beliefs including a belief in security by obscurity. As a result, security is too rarely considered a critical priority for embedded designs and many engineers are not yet aware of how the software they build can be exploited.

Developers of embedded systems software and connected device applications must begin integrating a security-first approach into their culture and processes if they are to keep ahead of what many researchers, analysts, and law enforcement agencies warn could be a new cybercrime wave.

MISCONCEPTION 2: SECURITY FEATURES EQUAL SECURE SOFTWARE

When software engineers and programmers think about security, they usually think about security features such as cryptographic ciphers and algorithms, passwords, and access control mechanisms. For an application to be secure, every aspect of the software must be secure, not just the components that explicitly address security.

In many cases, security exploits and vulnerabilities are not related to security features at all. A security feature can fail and compromise system protection in many ways, but there are usually many more ways in which poorly implemented non-security functions can be compromised and lead to an exploitable vulnerability. Security components are implemented with the understanding that they must operate correctly to maintain system security, but non-security features often fail to receive this same consideration, even though they are often just as critical to software security.

One of the most prominent examples of this is the failure to prevent buffer overflow, which remains the primary method used to exploit software by remotely injecting malicious code into a target. While control flow and memory vulnerabilities can take many forms, a buffer overflow is most often the result of an exceptionally simple bug that comes from a very common practice. Embedded software applications, which are typically power and memory constrained, operate on data received from both humans and machines. Applications allocate fixed-sized blocks of memory (buffers) to store this data as they work on it. A buffer overflow happens when more data is written to a buffer than the buffer can hold. The core bug that enables an attack to exploit this space limitation is writing more data to a buffer than the buffer has allocated space to handle. While this would seem like a problem that should be simple to avoid, it has its roots in the C programming language, which is notoriously vulnerable to these types of issues. This language and its sib-
lings, Objective-C and C++, by nature of their design make it easy to inadvertently insert these bugs when coding. The C language is old, widely used, and essential to embedded applications and devices due to its flexibility and lightweight design. At the same time, it doesn’t provide any built-in protection against accessing or overwriting data in any part of memory and doesn’t automatically check that data written to an array is within the boundaries of that array.

With this reality in mind, it clear that no amount of encryption can protect against buffer overflow exploits. Instead what is needed is an effective way to test code for memory handling issues that could be exploited so that they can be eliminated before the software is released.

**MISCONCEPTION 3: RELIABILITY AND SAFETY EQUAL SECURITY**

"Reliable software does what it is supposed to do. Secure software does what it is supposed to do, and nothing else."

Ivan Arce, CTO of Core Security Technologies

Ensuring that software functions according to requirements doesn’t mean the software is secure. Testing software functionality, even security functionality, misses many security problems because security problems usually aren’t violations of the requirements. Rather, security problems are frequently “unintended functionality” that causes the program to be insecure.

In software programming, mistakes are inevitable; producing quality software requires a systematic approach to finding bugs. This is why many organizations use functional testing, which involves running the software and comparing its output against an expected result. This is standard practice for organizations with mature quality assurance programs and rigorous bug elimination programs, but it can also lead to a false sense of confidence that they are addressing security issues.

The misconception is that security is just another component of software quality. Unfortunately, it is has been shown over time that it’s just not possible to maximize software security using function based quality assurance methods. In practice, most software quality efforts are geared toward testing program functionality. The purpose is to find the bugs that will cause the product to fail in the worst ways. Functional testing works well for making sure that under expected operating conditions software will work as intended, but this approach doesn’t work for finding security defects that aren’t related to security features. As long as organizations focus on software testing that only verifies the implementation of requirements, they will perpetuate an approach that is inadequate for finding security problems.
**MISCONCEPTION 4: DEFENSIVE PROGRAMMING GUARANTEES SECURITY**

While the term defensive programming is increasingly used with a security connotation, the term has been in use for some time and is often introduced in early programming courses. The term originally referred to the practice of coding with the mindset that errors are inevitable and that, at some point something will go wrong and lead to unexpected conditions within the program. This approach to improving software and source code takes into account:

- Improving overall quality by reducing the number of software bugs and problems
- Making the source code readable and understandable, so it can be approved in a code audit
- Making the software behave in a predictable manner despite unexpected inputs or user actions

Defensive programming makes sense in the context of introductory programming courses because most beginning programmers are usually their own worst enemy; for the most part, this approach serves to uncover coding constructs and logic errors made by the programmer. Defensive programming makes defects easier to find, easier to diagnose, and easier to fix.

This approach, however, does not guarantee secure software – although the concept of expecting anomalies is also a requirement of secure software development. Security requires a different perspective and must assume the need to thwart an attack and the presence of users intentionally trying to undermine the operation of the application and the system. So in contrast to defensive coding, which attempts to account for common kinds of mistakes (by either the developer or the user), software security is about creating applications that behave correctly even when malicious behavior is involved.

**SECURE DEVELOPMENT APPROACHES AND TESTING TOOLS**

Security must be incorporated into the software development lifecycle. To do this, organizations need to better understand security in the context of application design and development – a task that is much easier said than done. As development processes becomes more complex, the amount of code being written increases dramatically, and the need to reuse code becomes more acute, adding security concerns into the mix is often beyond the abilities of individuals and organizations to handle without technology to automate and manage better processes. A number of tools exist that can help improve security at different points of the development lifecycle. These include management tools and platforms that are used throughout development, and testing tools that are more applicable during specific phases.

**IDENTIFYING AND CORRECTING SECURITY FLAWS**

Classification systems for security defects have been in place for over 40 years and date back to the 1970s. Since the emergence of the Internet, however, security has become increasingly important and has received significant attention from academic researchers, government agencies, commercial vendors, and a growing number of standards organizations. Based on the work of these organizations, it is apparent that security defects share a significant number of common themes and patterns, making it possible to define and categorize them. Security flaws can be found both at the code level and the design level. Certain
types of flaws create vulnerabilities that can be identified without any knowledge of how the application is
designed and executed, while others require an explicit knowledge of the context in which the code is being
executed. The following tools have evolved to support security initiatives from both perspectives.

RISK AND THREAT MANAGEMENT TOOLS
Embedded applications are deployed in a wide variety of environments, targeted across a variety of attack
vectors to consider, and often developed using software acquired from multiple sources.

Automated risk and threat management tools analyze and refine threat data, automatically notify responsible
personnel, and proactively address threats before they impact an organization. These tools are often
used in projects where risk management processes are stringent and detailed traceability is required. Threat
management processes document geopolitical threats; consolidate vulnerability, malicious code, and patch
information from security intelligence providers; and capture vulnerability results from network and software
scanning technologies into a single threat management system.

Threat management tools can prove valuable to any team that needs to maintain detailed information about
their risk and security plan. Automating the management of security threats from early threat assessments
all the way through the device lifecycle enables organizations to organize and retain the details of each
threat, record the threat analysis, set security test and fix priorities, and enable end-to-end traceability.

As embedded components become more connected and interdependent, and as security requirements
become part of industry standards such as ISO/IEC 27002, threat management tools will become increas-
ingly essential.

APPLICATION LIFECYCLE MANAGEMENT TOOLS
Application Lifecycle Management (ALM) tools encompass a wide variety of software directed at improv-
ing software development productivity and quality while managing and limiting program and budget risks.
These tools typically support requirements and change management, as well as modeling and analysis,
test management, and product and portfolio management. ALM tools are employed to measure and moni-
tor compliance, quality, productivity, and security. Although they aren’t specifically designed as security
tools, incorporating security requirement management, secure design, and security testing into the ALM
process is critical. Put another way, these tools provide a way to manage the security design lifecycle as
part of the overall software development lifecycle.
SYSTEM SIMULATION

Secure design can be tested and system vulnerabilities identified early and often using system-wide, comprehensive simulation that supports testing a complete networked system, board, peripherals, and processor using simulation. The key benefit of system simulation is performing integration and test cycles early in the life-cycle under conditions that real hardware can’t meet and with the ability to arbitrarily inject faults at any part of the system.

ROBUSTNESS AND FUZZ TESTING TOOLS

Finding and fixing zero-day vulnerabilities and potential exploits is critical for embedded devices since it can often be very difficult, if not impossible, to update the software on these devices. Fuzz testing can help to identify these defects and is used to generate and test performance using out of range and erroneous data meant to overload the system. Fuzz testing tools are designed to send malformed data to the device to induce errors. These tools use knowledge of various communication protocols (including specialized embedded control protocols) to create incorrectly formed packets, bad data, and various edge cases. Often this data is enough to induce runtime errors in the device and thereby expose potential vulnerabilities. Fuzz testing can be applied well before the device is ready for release.

DYNAMIC CODE ANALYSIS TESTING TOOLS

Two types of analysis, which are complementary but often thought of as substitutes for each other are static and dynamic code analysis. Both are performed during source code reviews, however, static code analysis is done without executing any of the code and dynamic code analysis relies on examining how the code behaves during execution. Dynamic analysis is the testing of a program during runtime and examines how the application behaves when executed and how it interacts with other processes and the operating system itself. You can use dynamic analysis to measure code coverage and identify paths not taken during execution, which are therefore untested and may contain bugs or exploits. Dynamic analysis works by either integrating introspective code into an application at build time (the most common approach) or by employing emulation to understand the internal behavior of an application during execution.

STATIC CODE ANALYSIS TESTING TOOLS

Static analysis tools test source code to find defects, security vulnerabilities, and other programming issues. These tools can be applied to existing code bases as well as new source code as it is written by the developer. The key advantage of using static analysis is that security vulnerabilities are found as soon as the code is written – minimizing the cost and risk associated with fixing them. Also, vulnerabilities can be found in “hidden” parts of the code – code that is not executed during traditional testing. Static analysis tools can be run individually within each developer’s IDE and singularly on the complete project build providing a distributed and centralized analysis capability. Full-featured static analyzers can be configured to verify compliance with CERT C and other secure coding standards.

THE RELATIVE COST OF FIXING DEFECTS

One of the most well-known facts about software defects is that the longer they go undetected, the more expensive they are to fix. Although research differs on the exact ratios, the general rule is 1:10:100.

That is, if a defect costs one unit (for example one hour or one dollar) to fix in requirements and design, it costs 10 units to fix in system or acceptance testing and more than 100 units to fix in production. Sometimes the cost to fix a defect in production costs much more than 100 times the cost of fixing it in the requirements phase.
SUMMARY

Improving embedded software security is not optional. Any organization that ignores the warning signs of increased cyber attacks is likely to suffer economic consequences when it becomes the next target. The first step towards taking action and making the necessary investment is dispelling the illusion that security isn’t necessary or is already being addressed. Organizations must complement their functional testing focused on common usage patterns of the device with security testing that focuses on uncommon usage patterns, attack vectors, and attempts to break the system. Security is not the same as quality, nor is it the same as reliability. As a result security testing must consider edge cases, erroneous input, and misuse that does not occur under normal operating conditions.

Addressing security issues at the coding stage is more efficient and cost effective than addressing them later in development or after the product is released. In pursuing this goal, tools are essential to developer productivity. With the growing dependency on embedded applications, increased emphasis on security as well as quality and safety, and increased demand for more features with tighter schedules and budgets, automated tools capable of exposing security issues as early as possible are more vital now than ever before and essential to improving the security of embedded devices.

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.