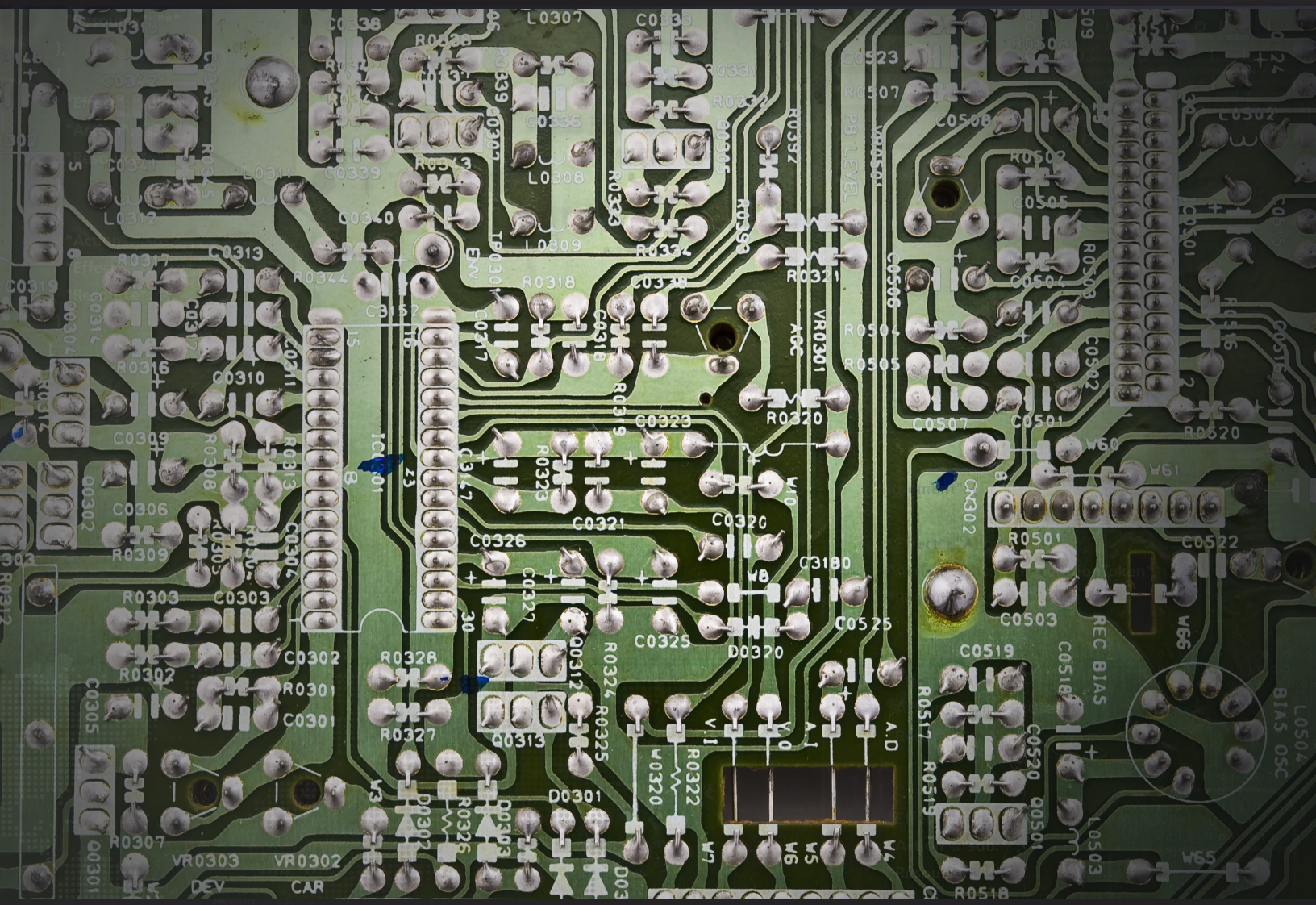
GRAMMATECH

EMBEDDED SOFTWARE DESIGN:

BEST PRACTICES FOR STATIC ANALYSIS TOOLS



TRUSTED LEADERS OF SOFTWARE ASSURANCE AND ADVANCED CYBER-SECURITY SOLUTIONS

WWW.GRAMMATECH.COM

INTRODUCTION

This paper reviews a number of the growing complexities that embedded software development teams are facing, including the proliferation of third-party code, increased pressures to develop secure code, and the challenges of multi-threaded applications. It highlights how static analysis tools such as GrammaTech's CodeSonar can detect defects caused by these complexities, early in the development lifecycle when they are most cost-effective and easy to eliminate.

The paper contains proof points for the value of early defect-detection in terms of accelerating time-to-market while helping teams build higher-quality, secure applications, and concludes with specific examples of how embedded development teams are finding and eliminating defects in their code.

BACKGROUND

Around the world, the adoption of embedded devices is growing at an unprecedented rate, with the global market for embedded systems expected to reach $194.27 billion by 2018 according to a recent report from analyst firm VDC Research. Beyond this sheer market momentum, software innovation continues to advance as embedded developers write increasingly sophisticated code for use in the aerospace, automotive, communications, industrial, medical, nuclear, rail and other fault-intolerant industries.

The quality and security of embedded applications is the gold standard for excellence in software development. This is because embedded applications perform functions that are essential to safety-critical activities, countless times per day. As such, embedded software – and the developers that write embedded code – must adhere to performance standards that exceed most other industries.

While enterprise applications perform many business-critical functions, embedded applications continue to proliferate in life-critical functions. So while the demands to build a reliable trading platform, enterprise HR application, or CRM system are high, the pressure to build a reliable pacemaker, automotive automatic braking system, or nuclear control system is extraordinary.

Therefore, on the rare occasions when these systems do fail, the repercussions are significant and often damaging. With today's voracious 24-hour news cycle, hungry for catastrophe and magnified by social media, any organization responsible for a device that fails or that is exploited by attackers suffers stiff penalties to reputation and bottom line.

The accelerating trend of networked devices – and the security risks that connectivity creates – demands new levels of rigor. Unlike the traditionally highly regulated industries, most embedded applications are created in a highly competitive, rapid turnaround commercial environment. Unfortunately, this can leave consumers at more risk to errors in code.

Given the downside of the risks enumerated above, it's reasonable to question why these failures still occur. The answer is simply that embedded applications are more difficult to build than most other types of software.

COMPLEXITY: THE NEW NORMAL FOR EMBEDDED SOFTWARE

Embedded developers confront daily pressures unlike coders in any other industry. While most development teams today are benefiting from cloud-based, homogeneous hardware and virtualization, embedded developers must build applications that deliver consistent, predictable performance across a growing array of heterogeneous hardware/processor configurations. This challenge occurs because embedded developers must deliver new features in the face of host of challenges, including the following.

Externally Developed Code

The use of outsourced and/or open source code is a common practice in software today. Unfortunately, externally produced code poses unique risks due to nested third-party supply chains, frequent inaccessibility to the library's source code, and the specter of malicious insiders surreptitiously planting exploitable security vulnerabilities.

Multi-Core Hardware

In order to take full advantage of today's multi-core CPUs, applications must be designed to be multi-threaded. Embedded software developers need to be aware of potential concurrency hazards that can cause erratic behavior or unpredictable crashes.

Security and Networking

Embedded systems are increasingly becoming network enabled, which exposes them to attacks that traditional embedded developers are not necessarily trained to mitigate. Whether it is code for network routers, medical devices, or home security systems, any device with network exposure becomes open to sophisticated cyber attacks.

Standards and Verification

Safety-critical software in avionics, automobiles, and consumer devices are subject to an increasing number of code quality and security standards, such as DO-178b/c, ISO 26262 and others. Coding standards such as MISRA C are increasingly recommended to help avoid the inherent hazards of the language. Regulatory agencies may require adherence to these standards, but even if not, they are widely recognized as best-practice.

Embedded Code Base Explosion

Embedded application code-bases are increasing at nearly 30% a year, according to industry estimates. The growing size of code bases means additional complexity for developers to deal with, and complexity, in turn, translates to a higher incidence of defects.

Shorter Cycles and Budget Pressure

Embedded development teams use a mix of waterfall and agile methodologies. Whichever coding philosophy a team believes in, every developer today is facing increased demand for faster cycles to add new features, respond to customer feedback, and fix known bugs. Yet, this 'need for speed' in development can come at the cost of writing quality code.

Risks of Embedded Languages

The most popular languages for embedded software are C and C++. These languages carry unique risks for developers because deficiencies and ambiguities in the language specifications

can give rise to undesirable and unexpected behavior during execution. Given the varying hardware/firmware environments that an embedded application may be required to operate in, it is difficult to test for unwanted behavior.

Due to these factors, embedded development teams have generally adopted formal coding and testing practices. In addition to standard QA testing, advanced automated tools are used to inspect the source code and performance of an application early in its development when defects are easier to eliminate, and provide sophisticated reports to make complying with standards most efficient.

Boston Scientific

Boston Scientific has more than 13,000 products worldwide. Among these offerings are many safety-critical medical devices, including implantable cardiac rhythm management products.

For years, the company has relied on GrammaTech CodeSonar to automate the analyses that were most manually intensive in the past, and whose reliability and repeatability were most important. The automated static analyses run in mere hours, compared to the person-weeks they took previously.

Boston Scientific also automated checking for a number of other potential quality and security issues early in the development lifecycle, including stack usage analysis and recursion identification. GrammaTech even collaborated with Boston Scientific to build customized analyses and additional reporting capabilities as extensions to CodeSonar.

> "The automated analysis provides a huge amount of leverage in a cost-effective way. It doesn't just free up engineers' time, it also means we can analyze our entire code base more often to ensure that our standards are continuously upheld, and to receive more frequent feedback on our code quality."
>
> — Gerald Rigdon
> *Software Engineering Fellow,*
> **Boston Scientific**

THE VALUE OF EARLY, AUTOMATED DEFECT DETECTION

So, for embedded development teams, whether developers are writing code that travels to Mars, or code that controls the brakes in a bus, or manages a pacemaker, identifying defects early in development is invaluable to the performance of their code. But how valuable, exactly, is early defect identification?

Of all the tools and strategies available to improve embedded software development processes, automated source/binary analysis offers some of the highest ROI. And, while automated detection of quality/security defects in embedded software is more efficient than manual processes, its greatest value may not just be what defects it finds, but *when* it finds them.

As evidence, consider the 2002 study by the National Institute of Standard Technology (NIST), which concluded that eliminating a single defect during development required an average of 5 hours, while defects found in a production environment took an average of 15 developer hours each to eliminate. Furthermore, a recent study by IBM's Systems Sciences Institute found an even greater disparity in the relative costs. The results are shown in the table on the following page (Figure 1).

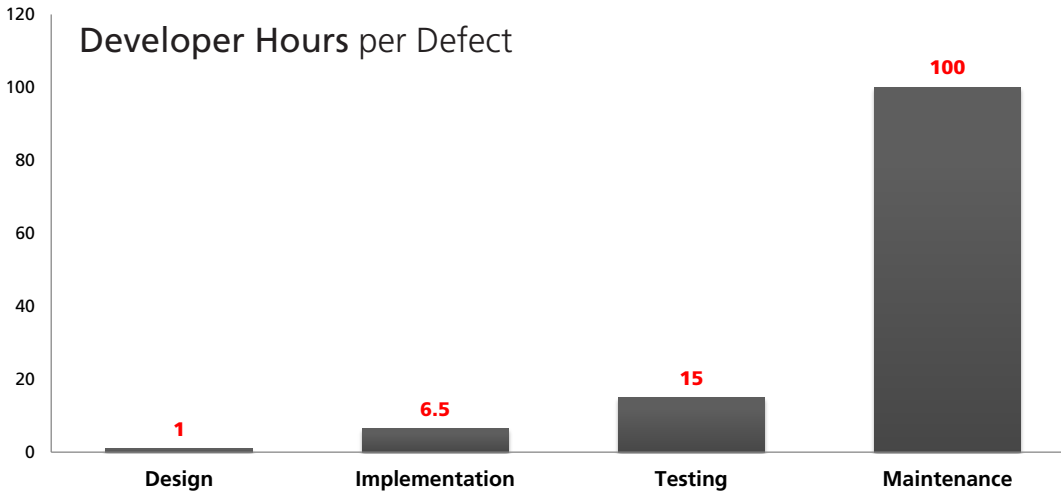


Figure 1. *Source: IBM Systems Sciences Institute*

With the proven value of early defect detection in the software development lifecycle and the array of pressures facing embedded development teams, automated code analysis proves to be one of the most cost-effective investments an organization can make to accelerate release cycles and improve developer productivity.

Working side-by-side with the world's leading device manufacturers and the sophisticated U.S. government agencies that build and test highly-secure, failure-intolerant code, GrammaTech's engineering team has a unique understanding of the rigor required of embedded software today.

AUTOMATED ANALYSIS REQUIREMENTS FOR EMBEDDED APPLICATIONS

GrammaTech's expertise in embedded software development stems largely from CodeSonar, the company's flagship static analysis product, which has processed over one billion lines of code deployed in a wide variety of failure-intolerant devices, such as NASA's Mars Curiosity Rover, Boston Scientific's implantable cardiac rhythm management products, or Sypris Electronics' encryption devices.

Based on the company's breadth and depth of embedded software expertise, GrammaTech recommends key capabilities that all development teams should require from their automated code analysis tools, including the recommendations that follow on the following pages. Each recommendation includes an example with real code that was analyzed in CodeSonar.

BINARY ANALYSIS

Although third-party code is used in almost every application, developers often lack the ability to analyze that code because it is not supplied in source-code form. Without the source, automated analysis tools can draw few useful conclusions about the quality and security of such code.

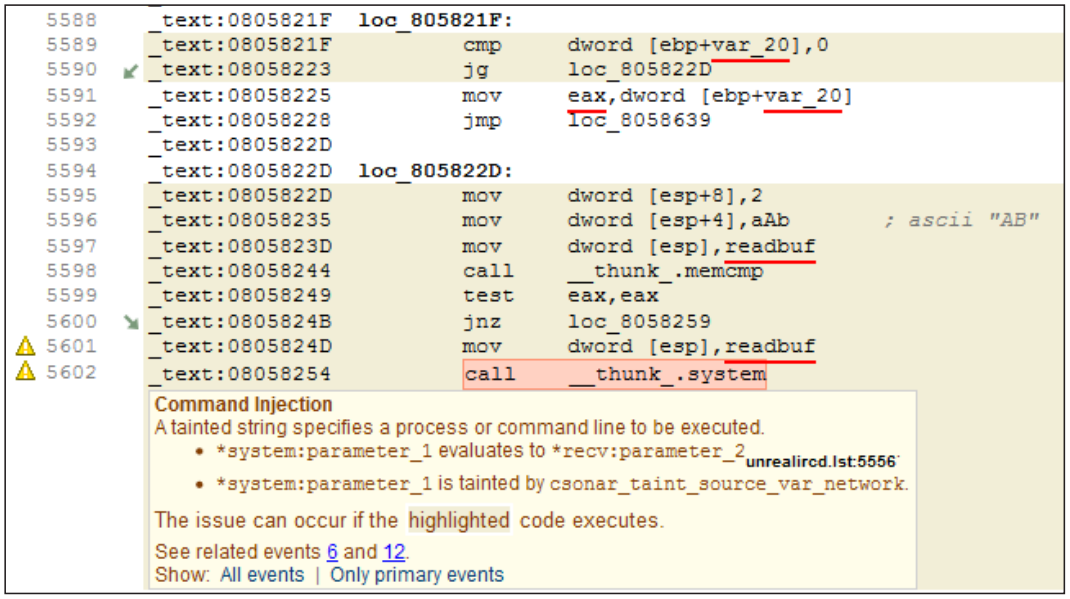


Figure 2.

VDC Research estimates that approximately 30% of code in embedded applications is third-party commercial software – for which source code is often unavailable. An automated tool that analyzes binaries will help eliminate this dangerous blind spot.

The code sample above (Figure 2) contains an example of a command injection vulnerability that was maliciously and surreptitiously inserted into a program named *UnrealIRCd* (see CVE-2010-2075). Line 5602 is a call to the `system()` function whose parameter comes from data read from a network connection. CodeSonar was able to find this defect in both the source code and the compiled code.

"NATIVE" SUPPORT FOR STANDARDS

The movement toward formal product development standards, such as MISRA, DO-178B/C, or ISO 26262, continues to grow worldwide. In addition, coding standards such as MISRA C are being increasingly used because they help to mitigate the hazards inherent in the use of the C language.

These standards are often used in combination across automotive, aerospace, medical device, industrial control, and other
embedded-intensive industries. Organizations in these industries must be equipped to identify not only the violations of superficial syntactic rules, but also serious bugs arising from undefined behavior, for example, as proscribed by the MISRA C:2012 standard. While some of these occurrences can be enumerated through testing, only the most advanced static-analysis tools are capable of finding the more subtle occurrences.

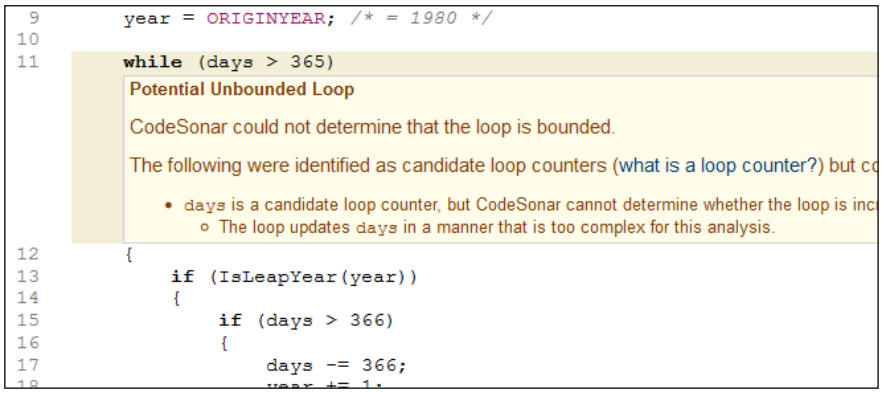


Figure 3.

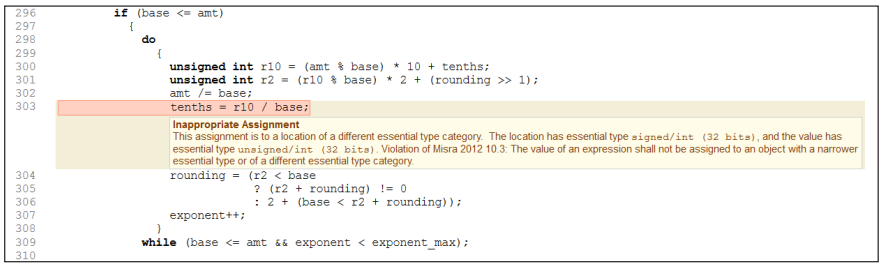


Figure 4.

Good examples of how failure to adhere to coding standards can result in serious defect making their way into production code are shown in Figures 3 and 4. The first code example (Figure 3) is a simplified version of the code that caused thousands of Microsoft Zune music players to fail on New Year's Eve 2008. This code contains an infinite loop; on the last day of a leap year, the value of `days` would become exactly 366 so the loop would not terminate. Because New Year's Eve 2008 was the first such day since the devices had been released, many of them were crippled by this defect. The defect could have been avoided if the authors had complied with a coding standard that required all loops be bounded.

The example in Figure 4 shows a type mismatch. The variable `tenths` is declared as a signed integer, but the arithmetic expression delivers an unsigned value. This is a violation of the stronger type-consistency rules defined by the MISRA C standard. Such type inconsistencies are prohibited by the standard because they can cause silent truncations and overflows that can lead to unexpected behavior.

INTEGRATED SECURITY

The rapid trend toward networking in embedded systems has created larger attack surfaces for malicious hackers to exploit. Code-injection attacks succeed when a malicious attacker feeds specially-crafted input data over an input channel to a program that fails to check that the data is well-formed and within reasonable bounds, and then passes it through to a sensitive part of the program. Programmers can defend against these vulnerabilities by treating input data as potentially hazardous (tainted) and validating it before the application is allowed to act on it.

Locating these potential exploits is a significant challenge because doing so requires manually tracing the flow of data from where it originates all the way through to where it is used.

A static analysis tool that can automatically track how potentially hazardous information can flow through the program can significantly reduce the time it takes to do an effective security assessment. Ultimately, ensuring that input data isn't tainted also reduces the risk and legal liability of compromised software reaching end-customers.

The example below (Figure 5) shows a format string injection vulnerability in *wu-ftpd* (CVE-2000-0573). This vulnerability allows a remote attacker to execute arbitrary code on the server by passing an exploit string to the SITE EXEC command of the FTP protocol. The red underlining indicates that the associated value is *tainted*. CodeSonar has detected this vulnerability and reported the path through the code that must be taken for the vulnerability to be exploited.

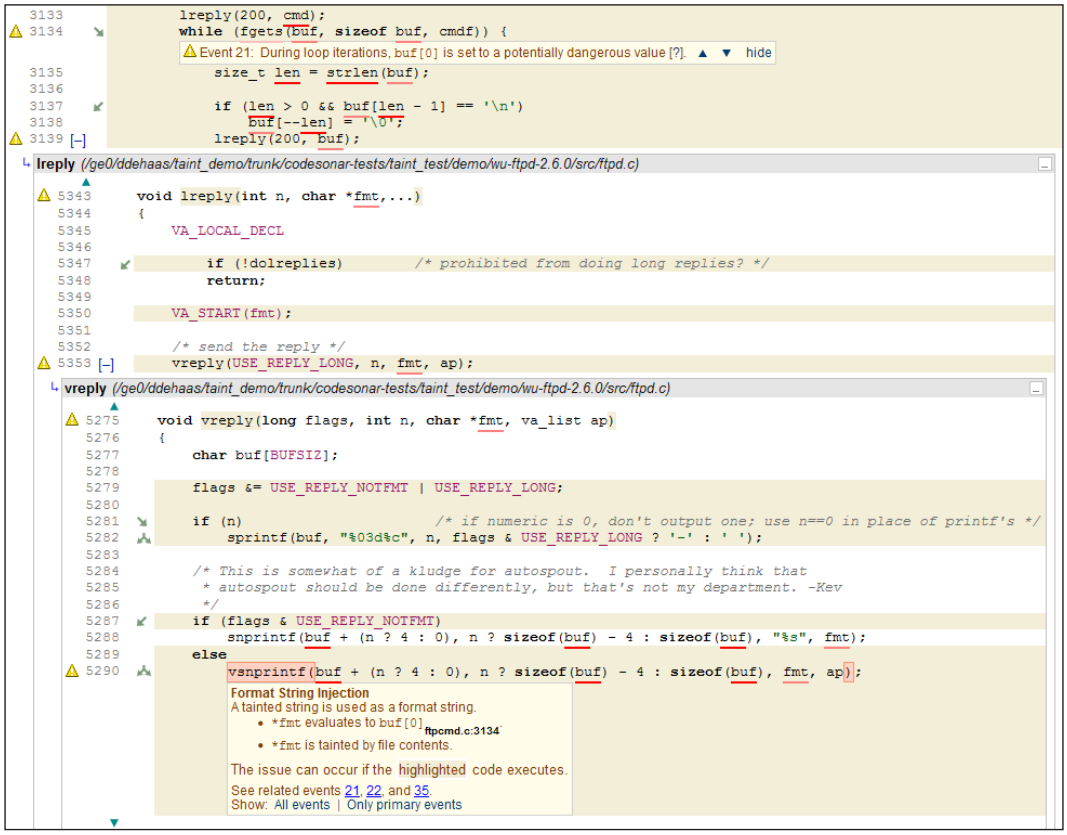


Figure 5.

CONCURRENCY CHECKING

In order to exploit the performance potential of multi-core processors, developers must write multi-threaded code; however, writing multi-threaded applications can introduce hard-to-find bugs because concurrent programming is inherently more difficult than single-threaded programming. Advanced static analysis solutions have been available to address concurrency problems for C and C++ programs, but until now, the industry has lacked a comprehensive solution for concurrent Java programs.

With 28% of embedded developers already using Java today, it is now the third most popular language for embedded systems, after C and C++. Development teams that do not successfully protect their code against errors like race conditions and deadlocks in C/C++ and Java will invariably experience product failures in the field. A good example can be found in the code below, in Figure 6. This code example shows an inconsistency in how the methods of a class

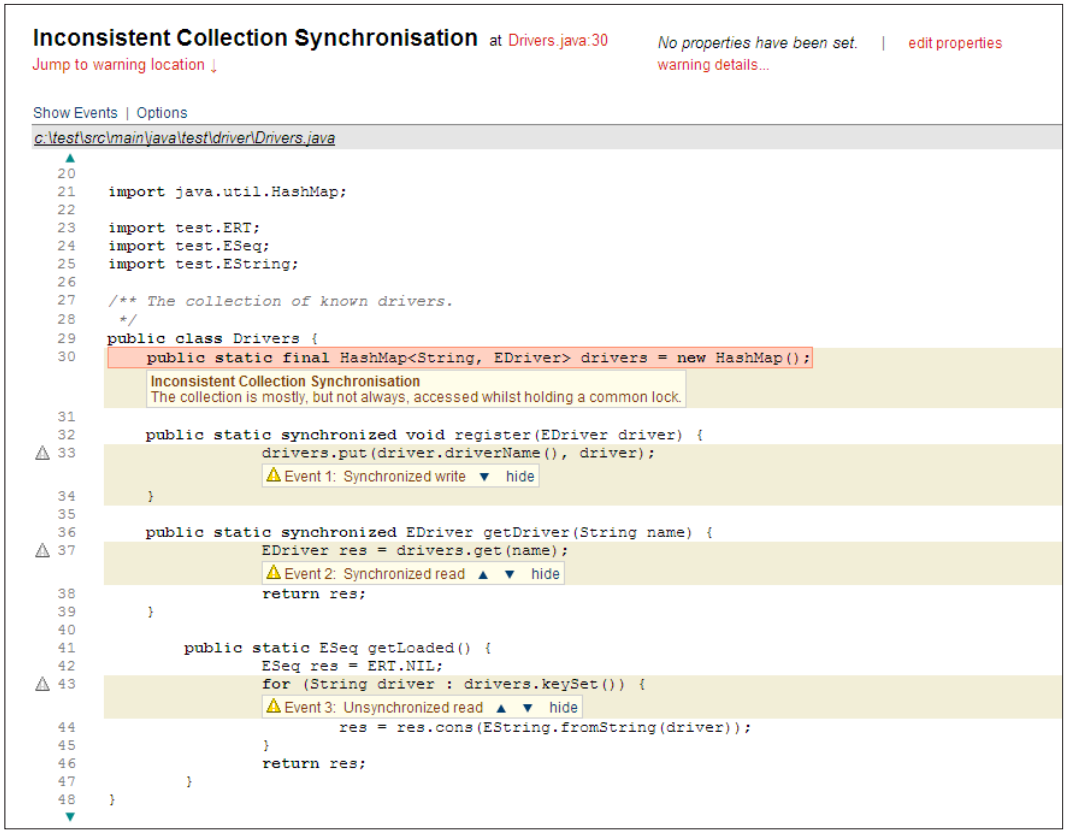


Figure 6.

access a field. In some of the classes the accesses are synchronized, but in others they are not. This kind of error is easy to overlook yet can lead to mysterious symptoms that are difficult to reproduce and diagnose.

COMPREHENSIVE DEPTH OF ANALYSIS

C and C++ are the most popular languages for embedded applications, but due to a number of inherent deficiencies, C and C++ programs are very susceptible to dangerous defects. When this characteristic is combined with an expanding variety of target hardware/firmware combinations, unpredictable behavior is hard to avoid. Selecting a tool that is able to dig deep into code bases while delivering a low false-positive rate is key to eliminating defects in code.

Tools that employ a unified dataflow and symbolic execution analysis to examine the computation of an entire program will find more potential defects and exploits, empowering embedded developers to deliver higher quality software. Additionally, teams that select a tool with advanced defect presentation capabilities such as visualization will be better equipped to understand the implications of code weaknesses in their embedded applications.

The code sample below (Figure 7) highlights a real defect found in code responsible for checking the validity of SSL credentials. In this instance, the static analysis tool is alerting the development team that the shaded section of code can never be reached. The error is that the 'goto' on line 35 is unconditional so the statements on line 36 through 39 are always unconditionally skipped. This is the kind of error that can easily creep in because of a bad cut-and-paste or an oversight while resolving a version control conflict.

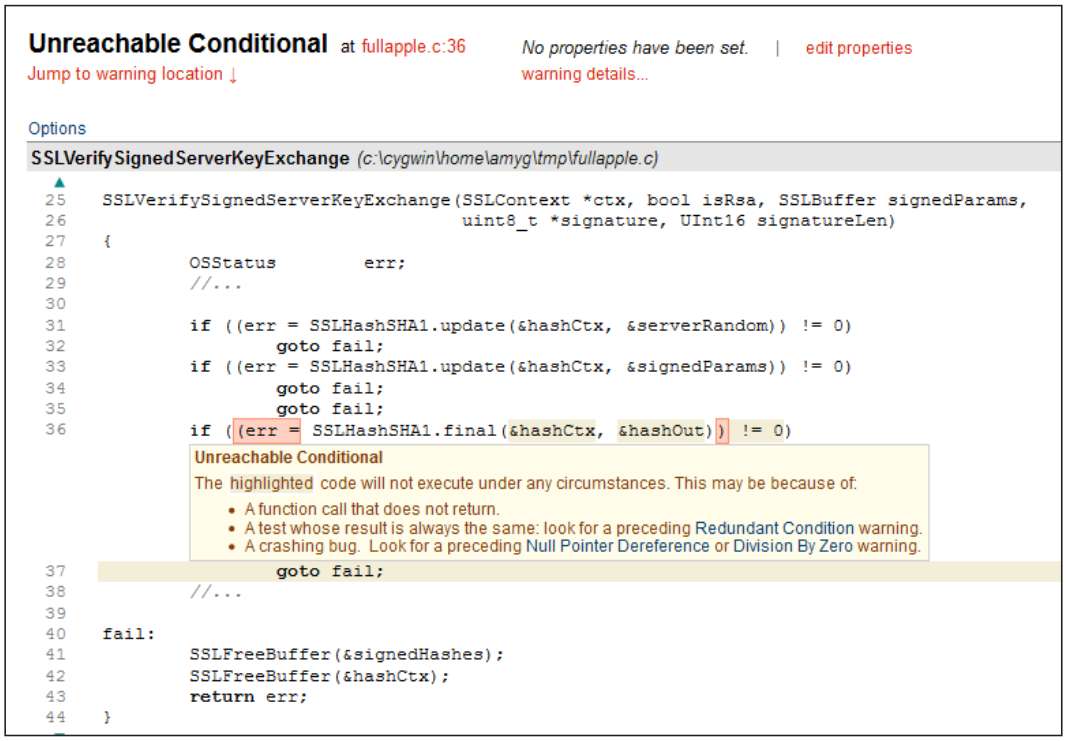


Figure 7.

CONCLUSION

Finding and eliminating defects early in the development lifecycle saves valuable developer time, accelerates release cycles, and produces code that is more secure and of higher quality.

Because the price of failure for embedded devices can be so high, development teams have historically been early adopters of advanced source code analysis solutions. Now, more than ever, as the stakes in the embedded industry continue to climb even higher, engineers must continually evaluate these automated analysis tools to ensure their success.

To succeed today and tomorrow, development teams need the most advanced static analysis tools to meet the challenges posed by new regulatory standards, to lessen the impact of the explosion of third-party code, and to manage the ubiquitous network-connected devices and multi-core processors.

After working with hundreds of commercial customers and many government agencies, including nearly all of those in the U.S. Department of Defense, GrammaTech's engineering team has developed an immense and highly specialized knowledge base regarding the most dangerous and hard-to-find defects in embedded software. This knowledge has fueled the research and development of CodeSonar, the industry's only static analysis tool engineered specifically for the rigors and complexities of code designed for embedded devices.

To learn more about how you can conquer the challenges facing embedded development teams, contact GrammaTech today for a complimentary consultation.

GrammaTech, Inc. is a leading developer of software-assurance tools and advanced cybersecurity solutions. GrammaTech helps organizations develop and release high quality software, free of harmful defects that cause system failures, enable data breaches, and increase corporate liabilities in today's connected world. GrammaTech's CodeSonar is used by embedded developers worldwide.