GRAMMATECH

White Paper **April 2014**

Eliminating Vulnerabilities in Third-Party Code with Binary Analysis

Background

Over the last few years, third-party code has moved from a minor factor in software development to a dominant force in the industry. It is now used throughout software development in all applications, from highly sensitive government and military applications to security-intensive consumer commerce and communications.

According to the latest report from VDC Research, the majority of software that runs on embedded devices is now developed by external sources, not in-house development teams. Some of this is open-source, but in embedded applications, nearly 30% of code is third-party commercial software – so the source is often unavailable. Such components include graphics and windowing toolkits, cryptography libraries, middleware, databases, and others.

As a result of this outsourcing, the behaviors of significant parts of applications are actually hidden from most of today's popular code analysis tools. Because third-party software is commonly delivered only in binary form, it cannot be examined with commercially available static source code analysis tools. Without access to the source code, these tools cannot fully account for the security consequences of executing the third-party code in the application.

Based on over 10 years of research, through collaboration with the University of Wisconsin and with support from the United States Navy, Air Force Research Labs (AFRL), and Defense Advanced Research Projects Agency (DARPA), GrammaTech has developed an advanced new capability that uses binary analysis to examine third-party code without requiring access to source code.

GrammaTech has integrated this binary analysis capability into their proven static analysis tool, CodeSonar, to create the first commercially-available binary analysis product. CodeSonar's binary analysis technology provides developers with the ability to evaluate, check, and inspect third-party code, all while reaping the benefits of advanced workflow options and management tools.

Common Third-Party Code Components

The use of third-party code has grown in popularity as more developers have started to build applications with a component-based architecture.

Some of the most common uses of third-party code include the following:

- **Communications** — Enabling an application to communicate via the Internet or wirelessly with other applications.
- **Databases** — Third-party software is used extensively to manage, optimize, monitor, and backup databases.
- **Standard Libraries** — These typically include definitions for commonly used algorithms, data structures, and mechanisms for input and output. Developers have become so accustomed to some of them that they forget the libraries are not part of the language itself.

The Dangers of Third Party Code

Development teams commonly turn to third-party software to incorporate particular functionality, such as communications or graphics, into their applications.

Cost, lack of local expertise, and unwillingness to "reinvent the wheel" are among the many sound reasons that organizations use third-party software, whether as components in their own products or as tools to support organizational activities. By outsourcing this development task, teams can focus more on the core functional capabilities of their software and dramatically accelerate time-to-market for their products.

When an organization releases software that includes third-party code, it becomes responsible for every line of code inside the application – including all of the third-party code.

A failure in a deployed product can result in significant losses. Producers who ship buggy code that subsequently fails can expect to lose reputation along with time and money. And even if bugs do not cause failures during normal use of the product, they may constitute exploitable security vulnerabilities. Fixing these defects can entail costly remediation.

Changes in development practices, ever-widening supply chains, and the rapid growth of code bases means it is now, more than ever, dangerous to assume that third-party code vendors have maintained and documented best-practices during their development processes. It is increasingly clear that blind trust in a producer is by no means sufficient to guarantee an acceptable level of risk.

Malicious entities can distribute counterfeit products, for example, to exploit the reputations of trusted producers. And genuine products themselves are not guaranteed to be risk-free: they can be tampered with in transit or sabotaged by malicious insiders within a trusted organization. Even if a genuine product is created by entirely trustworthy staff and delivered through a secure channel, vulnerabilities may still flow through from further up the supply chain.

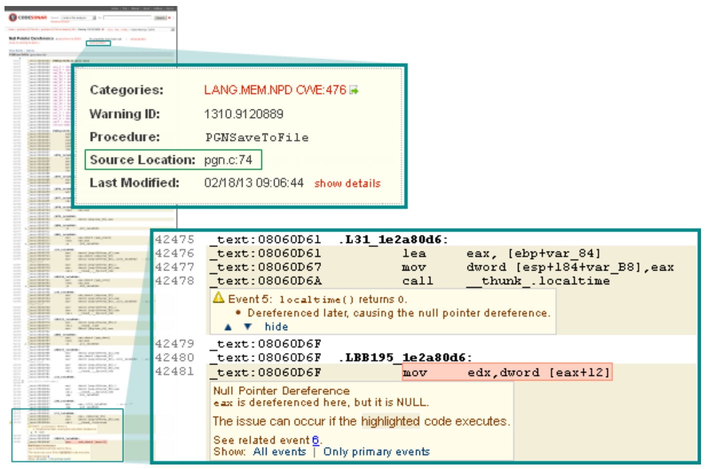
It is worth noting, also, that exploitable software vulnerabilities are not always caused by malicious interference. Errors that introduce zero-day exploitable buffer overruns, for instance, can arise from outdated design documents, misunderstandings about arcane language details, even typographical errors.

So what can development teams do to ensure greater safety in products that use third-party code?

Binary Analysis: An Innovation to Ensure Third-Party Code Safety

Instead of attempting to formulate and enforce code reliability requirements over the entire upstream portion of the supply chain, organizations can now employ a more practical approach. By leveraging binary analysis, organizations can focus on establishing trust in incoming software at the point of use, and in outgoing software at the point of dispatch.

This approach also permits organizations to consider a much broader range of software: products from new companies without established reputations, software obtained over unsecured networks, and even components whose provenance is completely unknown.



Categories: LANG.MEM.NPD CWE:476
Warning ID: 1310.9120889
Procedure: PGNSaveToFile
Source Location: pgn.c:74
Last Modified: 02/18/13 09:06:44 show details

```
_text:08060D61  .L31_1e2a80d6:
_text:08060D61          lea     eax, [ebp+var_84]
_text:08060D67          mov     dword [esp+184+var_B8],eax
_text:08060D6A          call    __thunk_.localtime
```

⚠ Event 5: `localtime()` returns 0.
- Dereferenced later, causing the null pointer dereference.

▲ ▼ hide

```
_text:08060D6F
_text:08060D6F  .LBB195_1e2a80d6:
_text:08060D6F          mov     edx,dword [eax+12]
```

Null Pointer Dereference
`eax` is dereferenced here, but it is NULL.

The issue can occur if the highlighted code executes.

See related event 6.
Show: All events | Only primary events

Figure 1. Here, CodeSonar has detected a null pointer dereference in an analyzed binary. Some associated build information was available, so CodeSonar was also able to determine the source location at which the dereference occurs.

CodeSonar can do static analysis on both binary libraries and binary executables. Binary executables can either include the symbol-table/debugging information ("unstripped") or not ("stripped"). Software producers may strip their binaries for a range of reasons, from benign (saving space), to proprietary (protecting trade secrets against reverse engineering), to hostile (obfuscating the code to hide malicious code).

Many tools are unable to extract useful information from stripped executables. CodeSonar, however, can analyze both stripped and unstripped executables.

Through this evolution in static code analysis, developers can inspect and evaluate all externally-produced code used in their applications. Binary analysis results can also be used to compare and contrast the relative safety of different third-party components, so teams can make the best possible decision when choosing components to include in their applications.

Integrated Analyses

For software developed entirely in-house, full source code is usually available for analysis. If third-party binary libraries are being used however, then the analysis must be able to analyze both source and binary simultaneously in an integrated fashion.

CodeSonar can analyze such code, so results take into account the flow of control and information between the source parts and the binary parts.

Recent Third-Party Code Failures

NETWORKING PROTOCOLS

In January 2013, the U.S. Department of Homeland Security issued a warning that third-party code embedded in approximately 50 million networked devices worldwide was vulnerable to infiltration by malicious hackers. UPnP enables networked devices to discover each other with Simple Service Discovery Protocol and establish network connections with a number of protocols, such as the Web's HTTP and Simple Object Access Protocol (SOAP).

This vulnerability impacted over 1,500 vendors and 6,900 products were identified as vulnerable, including products from trusted vendors such as Belkin, D-Link, Linksys, and Netgear. These vendors and millions of customers were exposed to remote attackers who could execute arbitrary code on their devices or execute a denial of service attack.

CONTROL SYSTEMS

Vulnerabilities in control systems (e.g., SCADA) can allow attackers to cause physical damage to equipment attached to those devices. Several presentations at the 2013 Black Hat conference reported on such instances. Scans revealed at least 90,000 vulnerable control systems connected to the Internet. For more information, read the article from technologyreview.com.

NETWORK SWITCHES

In 2012, the U.S. Industrial Cyber Emergency Response Team (ICS-CERT) reported a buffer overflow in a Siemens Ethernet switch. Remote attackers could exploit this by requesting a malformed URL, with possible consequences including device rebooting, denial of service, and purported "possible arbitrary code execution." As a result, Siemens was obligated to contact all customers that own the vulnerable product and ask them to patch the firmware on the switch.

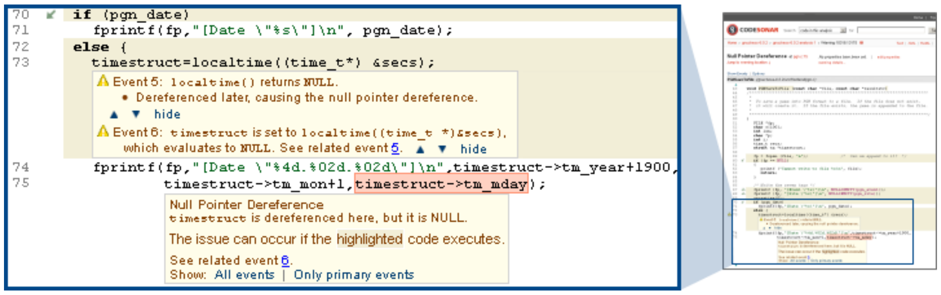
The analysis first creates a model of the entire program by first parsing the parts that are in source code, and disassembling the parts that are in machine code, then creating a single unified representation that captures the semantics of both parts in a consistent way. The analysis finds defects by traversing the model in an interprocedural path-sensitive manner, and looking for anomalies in the program state. When a warning is shown to a user, the path through the code to the point of error is shown, with important points along the path highlighted. When paths cross the machine code/source code boundary, the user can choose to drill down into the machine-code component, or can alternatively treat it as a black box with the relevant information projected onto the call site.

An additional advantage of analyzing binary code is that it represents exactly the software that will be executed by the hardware. Source code, by contrast, does not provide the whole story: the influence of the compiler must also be taken into account. Source code language definitions are full of ambiguities and inconsistencies. In such cases, the compiler is free to resolve these as it generates the machine code. Compiler optimizers frequently take advantage of these ambiguities. Thus, the semantics of the source code may even be different depending on the level of optimization used. Additionally, the compiler itself may contain flaws and generate incorrect code.

The example to the right shows a compiler-introduced error found during a 2002 security review at Microsoft. The compiler concluded that the memory was never accessed post-`memset()`, and so the `memset()` call could be removed, meaning that the cleartext password remained on the stack.

```
{
   char password[MAXLEN];
   ...
   ~~memset(password,’\0’,len);~~
}
```

When analyzing binary executables, on the other hand, all of these compiler effects have already manifested, so the analysis has much higher fidelity.



```
if (pgn_date)
   fprintf(fp,"[Date \"%s\"]\n", pgn_date);
else {
  timestruct=localtime((time_t*) &secs);
```
⚠ Event 5: `localtime()` returns `NULL`.
- Dereferenced later, causing the null pointer dereference.

▲ ▼ hide

⚠ Event 6: `timestruct` is set to `localtime((time_t *)&secs)`, which evaluates to `NULL`. See related event 5. ▲ ▼ hide

```
   fprintf(fp,"[Date \"%4d.%02d.%02d\"]\n",timestruct->tm_year+1900,
              timestruct->tm_mon+1,timestruct->tm_mday);
```
Null Pointer Dereference
`timestruct` is dereferenced here, but it is NULL.

The issue can occur if the highlighted code executes.

See related event 6.
Show: All events | Only primary events

Figure 2. This is the CodeSonar warning report for the source manifestation of the bug from Figure 1. CodeSonar can handle projects where both machine and source code are available for some component, but only source code is available for other components.

When Source Code Analysis Looks Comparatively Ordinary

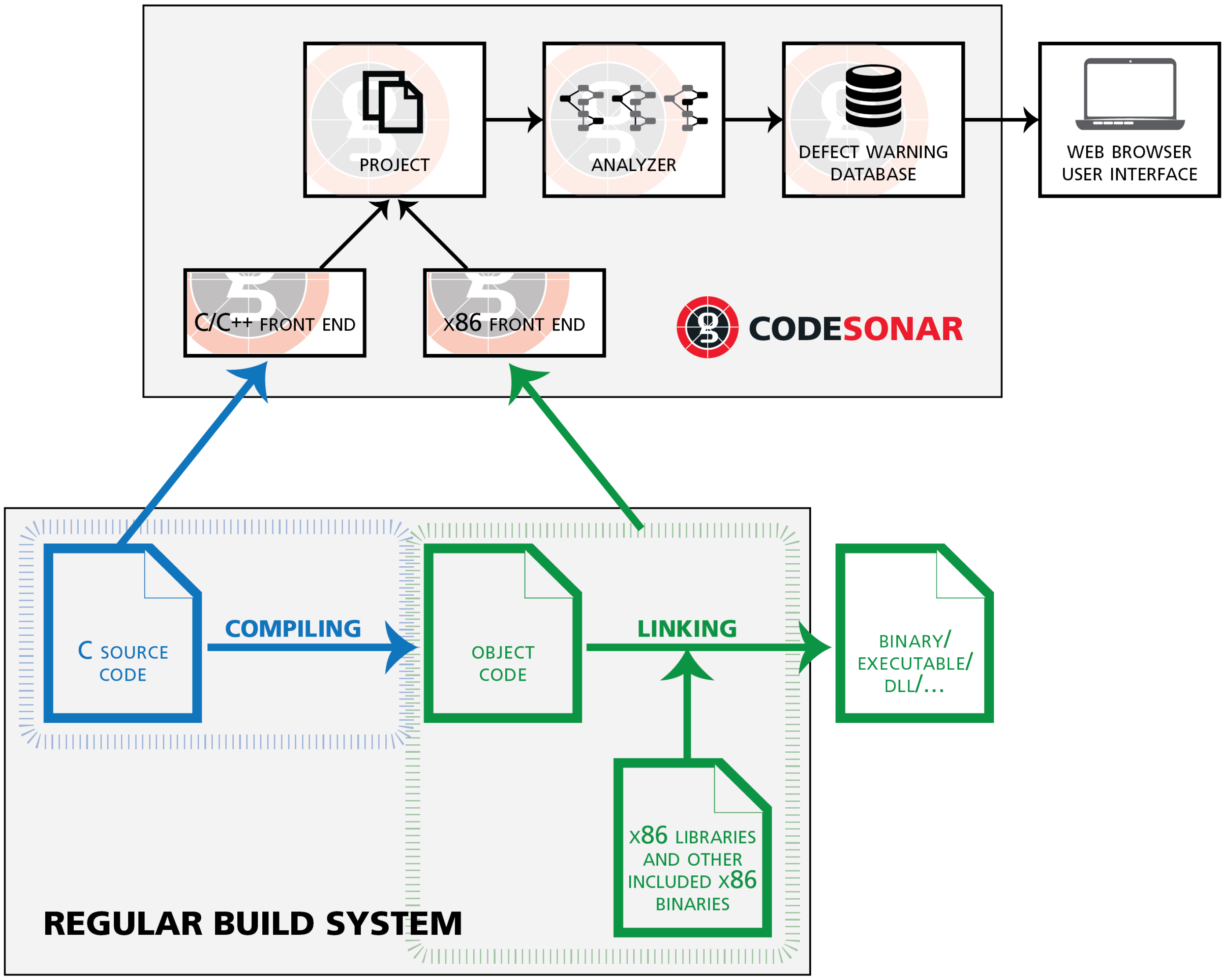
Analyzing machine code requires different techniques than those used for analyzing source code. Specifically, analyzing source code requires the availability of certain information:

- » A control-flow graph (CFG), or interprocedural CFG (ICFG)
- » A call-graph
- » A set of variables, split into disjoint sets of local and global variables
- » A set of non-overlapping procedures
- » Type information
- » Points-to information or alias information

When analyzing machine code, however, much of this information cannot be easily extracted, so developing an automated binary analysis tool requires an entirely different set of analysis techniques.

On the other hand, valuable information can be understood at the machine-code level that is not available for source code analysis. For example, source code analysis tools usually assume that the area of memory beyond the top-of-stack is not part of the execution state. With this assumption, the tool will be unaware of a malicious program’s use of that part of memory to store information. With the addition of binary analysis, it is possible to track the state of the area beyond the top-of-stack, for enhanced protection.

The following diagram demonstrates how binary analysis expands the code analysis footprint of an application.



Analyzing binaries with automated static analysis – in which run-time properties of programs are computed without actually executing the programs – has some important advantages over other methods for security assessment.

Unlike manual inspection, it scales readily to the size and complexity of modern software. Unlike testing, which can only ever cover a tiny portion of the possible execution cases, static analysis approaches coverage of all possible executions. Unlike dynamic analysis, which examines software as it runs, static analysis does not involve executing software. Inspection, testing, and other dynamic analyses can be helpful adjuncts to static analysis, but they cannot replace it.

Best Practices for Securing Third-Party Code

Used early in the development lifecycle, an automated binary analysis tool will help development teams select the safest components to include in their completed applications. Additionally, when using third-party code to build an application, development teams should follow other third-party code best practices, as described below.

Legal requirements: When contracting with a third-party software vendor, specify in the contract itself the security limitations your development team is willing to accept, as well as what constitutes a transference of liability to the third-party vendor.

Reporting transparency: Require that third-party vendors, in lieu of sharing their source code, share reports from their own use of automated software analysis tools and manual testing evaluations.

Coding standards: Discuss compliance of coding standards with third-party software vendors to understand what inconsistencies in their code may generate potential exploits, and to gain better knowledge of your vendor's coding process.

Communication: After analyzing your final application with binary analysis, work with your third-party vendors to improve the overall security of their code, and, by extension, your application as well.

Conclusion

Leveraging binary analysis to test and inspect the executables of third-party code will help developers build safer applications and instill greater confidence in the companies or government agencies that rely on the dependability and security of their software.

CodeSonar's binary analysis capability empowers developers with a new depth of understanding about how safe and secure applications truly are. Although acceptance testing of third-party components such as libraries remains important, developers are now able to build even safer applications by analyzing these components in the context in which they are being used.

Further, extending safety and security efforts into third-party code has important business benefits. It can accelerate development cycles, improve the security of software, and ultimately increase customer satisfaction.

Adding binary analysis to the development process allows developers to test a more holistic representation of their final application, which helps organizations deliver more trusted applications to customers and eliminate potential liabilities due to vulnerable third-party code. ■

GrammaTech's Binary Analysis Research

GrammaTech began researching and developing machine-code analysis and vulnerability detection tools in 2001.

GrammaTech's world-class binary research team is led by Dr. Alexey Loginov, Associate Vice President of Binary Analysis Technologies. The team's extensive experience includes over 60 person-years of research on machine-code analysis, and these scientists are responsible for many firsts in the field of machine-code analysis:

- The first model checkers for machine code and self-modifying code.
- The first property checker that can be applied to a stripped device driver to check that it conforms to an API-usage rule.
- The first tool for understanding the flow of values through an executable's variables and dynamically allocated memory objects.

References:

The Global Market for Automated Test and Verification Tools, VDC Research, 2013

WYSINWYX: What You See Is Not What You eXecute, IFIP Working Conference on Verified Software: Theories, Tools, Experiments (VSTTE), Balakrishnan, G., Reps, T., Melski, D., and Teitelbaum, T., 2005. Zurich, Switzerland: Springer.

Investigative Report on the U.S. National Security Issues Posed by Chinese Telecommunications Companies Huawei and ZTE, House Permanent Select Committee on Intelligence 112th Congress 2nd Sess., . 2012, USGPO.

About GrammaTech

GrammaTech's tools are used by software developers worldwide, spanning a myriad of embedded software industries including avionics, government, medical, military, industrial control, and other applications where reliability and security are paramount. Originally spun out of Cornell's computer science labs, GrammaTech is now both a leading research center for software security and a commercial vendor of software-assurance tools and advanced cyber-security solutions. With both static and dynamic analysis tools that analyze source code as well as binary executables, GrammaTech continues to advance the science of superior software analysis, providing technology for developers to produce safer software. To learn more about GrammaTech, visit www.grammatech.com.