Static Analysis of Machine Code for Supply-Chain Risk Management

Paul Anderson
Alexey Loginov
GrammaTech, Inc.
Ithaca, NY. USA.
{paul,alexey}@grammatech.com

Abstract— This paper discusses the product-oriented approach to software supply-chain risk management; a determination of the trustworthiness of software applications, or the relative trustworthiness among a set of software applications, based on automated analysis and inspection of their actual binary machine code. The system, named CodeSonar™ for binaries, is a static-analysis tool that can find security vulnerabilities in stripped and optimized executables. It is built as an extension to a successful product for analyzing source code, so it is also capable of analyzing source and machine code simultaneously. It can find defects such as buffer overruns, null pointer dereferences, resource leaks, and uninitialized variables.

Keywords-component; static analysis; security vulnerabilities; machine code; supply-chain risk management.

I. INTRODUCTION

Cybersecurity is one of the most serious economic and national security challenges we face as a nation, and Supply-Chain Risk Management (SCRM) is a priority of the US Government’s Comprehensive National Cybersecurity Initiative. Software is unavoidably global. Even organizations that consider themselves purely domestic are certain to use software that originates, in whole or in part, not only outside the organization but outside the country. SCRM therefore necessarily includes determining whether the software on which we rely is trustworthy.

Security vulnerabilities that put the supply chain at risk may have been unintentionally introduced or, more insidiously, they could have been deliberately introduced by a malicious insider.

Software producers relying on third-party components must establish trust in those components before delivery to their customers; software consumers acquiring applications from the global marketplace must establish trust in those products, ideally before adoption but if necessary after adoption during an audit. To establish trust in software, one may question whether the software originates from trustworthy sources whose development processes are themselves trustworthy, or one may ask whether the software product itself is trustworthy based on its own intrinsic properties. The former approach may defend against unintentional vulnerabilities due to low-quality software processes, but it is inherently passive. There are good reasons to be suspicious of security analyses done by vendors.

A recent report to the House Permanent Select Committee on Intelligence stated: “Vendors financing their own security evaluations create conflicts of interest that lead to skepticism about the independence and rigor of the result” [2].

The alternative to passively putting full trust in the supplier is to establish trust in the software artifacts themselves directly by using analysis tools to scrutinize them in detail. This approach is superior in that it avoids the vendor’s conflict of interest, and because it is much better for defending against the risk of insider attack.

In this paper we describe CodeSonar for binaries, a static-analysis tool capable of finding security vulnerabilities in stripped and optimized executables and libraries. Because it is based on an existing tool for source code, it is capable of analyzing both source and binary representations simultaneously.

The remainder of this paper is structured as follows.

Section II gives the motivation for the tool. Section III describes how it works. Section IV gives an example showing how it can find the same defect in both source and binary. Section V discusses some use cases. Finally, Section VI summarizes our conclusions.

II. MOTIVATION

The analysis of machine code has some advantages over source code analysis. Computers do not execute source code — they execute machine code, and there are certain classes of vulnerability that can only be detected by examining the machine code. The source code of a program may look benign, but an optimizing compiler can interpret language ambiguities in such a way that it will use perfectly legal transformations that introduce problematic machine code. As compilers are themselves fairly complex software applications, they can contain their own bugs that can cause incorrect code to be generated. Finally, analysis of machine code is one of the few ways that a Thompson-style attack [5] can be detected.

There are many static-analysis tools for analyzing source code that are available but even so, there are several good reasons why it is important to be able to analyze binaries as well.
Necessity. The source code may simply be unavailable, such as with most COTS software products. Even in a software development scenario it is very rare that all the software is available in source-code form. Most applications make use of binary libraries such as those that are part of the host platform, or those that are licensed from third-party vendors.

Convenience. Even when all the source code is available, it can be infeasible to configure the tool-chain to produce an executable that precisely matches the version of interest.

Breadth. A machine-code analysis tool can in principle work on code that has been compiled from any source language.

Fidelity. Analysis of machine code may be more accurate than analyzing the source code. One aspect of this is that some properties of the program in binary form may be impossible to detect in the source code. This is sometimes referred to as the WYSINWYX effect (What You See Is Not What You eXecute) [1]. A good illustration of this is a security vulnerability that was found by Microsoft during a security audit [3]. The program in question was required to read a password from the user into a buffer, then to validate it. The programmers who implemented this function chose to place the buffer on the stack, and after the text of the password was processed, adhering to the secure-coding guideline “minimize the lifetime of sensitive data”, they called a function to zero out the characters of the buffer. The code in question looked roughly as follows:

```c
char password[LEN];
...  
memset(password, '\0', LEN);
return;
```

To most programmers, this code looks perfectly reasonable, and indeed it would be, if it had executed as written. The problem was that the code did not execute as intended. Instead, the optimizing compiler noticed that the buffer named `password` was not used before going out of scope, concluded that the call to `memset` was a useless memory write, and generated code with the call elided. Consequently, the plain text of the password remained on the program’s stack, thereby confounding the intent of the programmer.

The example shown was inadvertent, but it is unreasonable to imagine that a clever malicious insider who knows about how an optimizing compiler works could exploit that knowledge to insert a vulnerability that is not apparent to someone looking only at the source code.

A second aspect of fidelity is that source-code analysis tools are at a disadvantage compared to binary analysis tools in that if they are to be sound, they must analyze the code in all possible ways in which it might be interpreted by a compiler. Source-language specifications (especially C and C++) are riddled with ambiguities and there are many instances where the specification leaves important choices to the compiler writer. For example, in C the order of evaluation of operands to a function is unspecified. To be sound, a source-code tool must analyze all possible combinations of orders, which can make the analysis very time consuming. This is a non-issue with a binary analysis tool because it operates on the exact output of the compiler after all those choices have been made.

Finally, compilers themselves are complex programs and may have bugs such that the generated code is just wrong. A source analysis tool has little hope of being able to find defects that are a consequence of incorrect code generation.

III. HOW CODESONAR FOR BINARIES WORKS

GrammaTech’s long-term vision has been to develop a retargetable machine-code analysis infrastructure suitable for the development of tools for software assurance, reverse engineering, rewriting, and producibility. CodeSonar for binaries is the culmination of this work combined with our pre-existing product for analyzing source code.

A. Program Model

CodeSonar works by creating a whole-program model of the program. This model is organized by subprograms and consists of representations such as the call graph, the control-flow graph (CFGs), abstract syntax trees (ASTs), and symbol tables with type information. A variety of techniques are used to traverse the model, looking for properties of interest. In our experience, the kind of analysis that finds the most interesting properties (i.e., bugs such as buffer overruns, null pointer dereferences, etc.) is a whole-program path-sensitive analysis. Roughly speaking, it does a symbolic execution, modeling the state of the program as a set of equations that encode properties of abstract memory locations and how they relate. When an anomaly is found, the analysis generates a warning that shows the path to the point where the anomaly occurs (e.g., where the buffer overrun happens) with additional points of interest along the path highlighted. Results are reported to a server and a web-client-based user interface is used to present them to end users.

Creating the model from the source-code representation of a program is relatively straightforward — the code is parsed and converted into ASTs and CFGs using standard compilation techniques. In contrast, the creation of a model from a binary representation is much more complicated. The process begins by using an off-the-shelf disassembler (IDA Pro) to extract a first approximation, and proceeds by refining the model. Some of the challenges in doing this are the following:

Distinguishing code from data. In source code, there is a clear distinction between instructions and data, but in machine code the boundary is less clear. Code and data can be intermingled arbitrarily. One way to find all the code is to find all locations in the machine code where control is passed to a new address. Because indirect jumps and calls are possible, finding all such locations amounts to a whole-program points-to analysis, which is undecidable in general. To make matters worse, the same memory location can even be treated as both code and data, such as with self-modifying code. Consequently, the question of whether a given location contains code or data is undecidable, so the best that can be done is to make approximations. CodeSonar begins by using the information from IDA Pro, and employs additional heuristics to improve the model.
Subprogram identification. In source code, subprograms are explicitly defined and have a single entry point; even if they have multiple exit points, the textual bounds of the subprogram are easy to find. Again, in machine code there is no strong boundary. Even if the code has been compiled from a high-level language where the boundaries are clear, an optimizing compiler can shuffle instructions around and the boundary can become blurred.

Library identification. With source code, calls to libraries are explicit and easy to identify. With machine code, the code for the libraries themselves may be loaded through dynamic linking, and calls to those libraries show up as indirect calls through a table of thunks that are resolved at load time.

Recovery of variables and parameters with their sizes and types. With source code analysis, variables are explicitly declared in terms of a type and a size, aliasing relations can be derived relatively easily from the code, and there are rules for type conversions. Although a weakly-typed language such as C can make it difficult for an analysis to track types, such constructs do not generally prevent analyses from producing useful results.

With machine code, there are no types and data can reside in registers that are used for multiple purposes, or in memory that is essentially a homogeneous vector of un-typed values. If a machine-code program is known to respect a particular application binary interface (ABI), then the convention for passing parameters to a procedure is known, but the sizes and types of those parameters may be difficult to recover.

CodeSonar’s analysis works best when it has knowledge of the types and sizes of variables and parameters, so it attempts to derive this information by looking for patterns in how the memory is accessed. Dereferences of a value indicate that its type is probably a pointer. If the dereference also uses an offset value with a fixed stride within a loop, then the type of the value is likely to be an array.

Resolution of indirect jumps and calls. Although it is fairly common for source programs to use indirect function calls, indirect jumps are not even possible in most popular high-level programming languages. In contrast, optimizing compilers often generate code that uses indirect jumps. A good example is a jump table generated from a switch statement. Creating an accurate CFG for a procedure requires that the targets of the jumps be determined, but this can be difficult. CodeSonar can use a variant of value-set analysis VSA [1] to resolve some indirect targets.

CodeSonar employs several techniques to overcome these difficulties. These include ….

B. Checks

CodeSonar finds three categories of defects. The first and most important category is defects that are caused by the program violating a fundamental rule of the source-language run-time. These are the category of defects that is likely to cause the program to crash or stray into undefined behavior territory. This includes memory access errors such as buffer overruns, and null pointer dereferences, as well as arithmetic errors such as dividing by zero.

The second category is comprised of those defects that stem from the misuse of an API. For example, a program that closes a file descriptor twice is one that is violating the rules for using the C library. Resource allocation errors, such as memory leaks, or using memory after it has been freed, fall into this category. These may or may not cause crashes. At best, they leave the program in an inconsistent state.

The third category consists of instances of code that are not necessarily defects, but indicate that the programmer may have misunderstood something important, or that something unexpected might happen. For example, an unsafe cast of a value into a different type. Or, a condition that is either always true or always false. These kind of superficial flaws correlate reasonably well with real defects.

CodeSonar for binaries is oriented towards finding defects in the first two of these categories. Defects in the third category are typically undetectable in the binaries because the compiler does not translate such defects, or because they are optimized away entirely.

C. Retargetability

An important feature of CodeSonar for binaries is that it can be retargeted for different instruction set architectures. It uses a language named TSL (Transformer Specification Language) [4] to specify the semantics of the instruction set, and an architecture-independent way to specify analyses. The TSL compiler transforms these specifications into a set of concrete types and code that form the core of the implementation of the analysis. Consequently the analysis can be retargeted to a new processor with much less effort than it would take to develop a new analyzer from scratch.
IV. EXAMPLE

Figure 1 shows an example warning in the source code of a program. In this example, there is a call to a library function named `localtime` that the programmer expected to return a valid pointer. The analysis tool has correctly pointed out that if the function encounters an error, then the pointer will be NULL and the dereference on line 75 will trigger an exception.

Figure 2 illustrates that CodeSonar for binaries has detected the same defect in the compiled representation of the program. The report is explained to the user in terms of a path through the disassembled machine code. Note that because this executable was compiled with debugging information, that the analysis is able to report the file and line number in the source code. Note that the debugging information is only used for reporting the defect, and never by the analysis itself.

V. USE CASES

There are many examples of use cases where a tool that analyzes binaries can yield significant advantages. This section describes some of these.

A. Augmentation of Source-code Analysis

It is very rare that all the source code for an application is available to be analyzed. Applications almost always use libraries (e.g., the C library) that are only available in binary form. When a source-code analysis tools encounters a call to a function that is not available in source code, it must treat that call in a way that does not compromise the analysis of the remainder of the code. A treatment that is too conservative, e.g., one that assumes that the call can do essentially anything, may be sound, but is sure to yield too many false positives to be useful. On the other hand, a treatment that is too liberal can yield false negatives. For commonly used libraries, tool vendors ship library models that capture the important aspects of the functions. If models are not available, then most analyses will err on the liberal side. For example, CodeSonar treats such calls as essentially no-ops, and the return value is tagged as unknown.

CodeSonar for binaries is capable of analyzing source- and machine-code forms simultaneously. Thus when the source analysis sees a call to a function defined in the binary, it can analyze the actual code. The result is a more accurate analysis.

Note that in this case the user does not have to look at the disassembled machine code in order to gain an advantage to using the tool. This is important during the development process because reading assembly-language programs is a specialized skill and many developers do not have the requisite training.

B. Tool vs. Service Model

An alternative to using a tool for analyzing binary code is to use a third-party code analysis service to find risky properties of the code. Typically, the users of the service first upload their code to the provider. The provider will analyze the code, typically using a combination of automated and manual techniques, interpret the results, and then send a report back to the users after a few days.

This is a reasonable option in some situations, but is ill-suited for many organizations. One of the most significant disadvantages of the service model is the requirement that users upload the code to the service provider. In some cases intellectual property requirements make it impossible to do this. Developers who work on classified projects must keep their code on secure systems, and are disqualified from using any remote code-inspection service.

Another downside of the service model is that the results of the analysis are interpreted and judged by people who were not involved in the development of the code. Their lack of familiarity with the software may mean they will misinterpret the code or the requirements, so important results may be misjudged. The results of a static-analysis tool are best judged by those who wrote the code under analysis.

The turnaround time may also be a disadvantage for some users of a service model. A delay of a few days means that the analysis results may be stale by the time they are communicated to the developers. In contrast, a tool can deliver analysis results within a few hours at worst, and within a few minutes if the analysis is done in an incremental fashion. The best way to get value from a static-analysis tool is to integrate it with the development process so that the results are delivered...
in reasonable time to those that need to act on them, and so that many developers can collaborate to address the issues raised. This kind of integration is difficult if not impossible to achieve within the service model.

C. Insider Threat

A nation state wishing to mount a cyber-attack on an adversary will use any means at its disposal to make such an attack easy and effective. This may include planting their own agents in software development organizations, or even coercing existing developers into doing their bidding. Such an insider could deliberately plant security vulnerabilities in software to make it possible to mount a successful attack at some point later in time. It is very tricky for a software development organization to defend against insider attack of this form.

The inserted vulnerability could be a defect such as a buffer overrun, but it could also take the form of a backdoor that would allow an attack to later bypass or override the normal authentication mechanism, or a time bomb that is set to go off and wreak havoc at some point in the future.

A successful attack of this kind is likely to have the following properties:

1. The vulnerability will be hard to detect with automatic or manual techniques.
2. The agent may wish to claim plausible deniability in the event that the vulnerability is detected. Thus it should look as if it were an inadvertent mistake.

One way to make the vulnerability difficult to detect is to exploit WYSINWYX effects such as those described earlier.

The ultimate example of an insider attack that exploits the WYSINWYX effect is the Thompson attack. In his famous address marking his acceptance of the Turing award in 1984 [5], Ken Thompson described a mechanism that has come to bear his name. The essence of this idea is that the compiler has been modified to insert a vulnerability in a program that it compiles (e.g., a backdoor in a login program). If the compiler is being used to compile itself, then it inserts the vulnerability-creating code into the executable. Once the compiler has been used to compile itself, the source code that inserts the vulnerability can be removed. The interesting consequences of this are that the vulnerability is self-propagating — all future versions of the compiler will have it, and that it is completely undetectable with any source-code analysis tool. The only way to detect it is with a tool that can analyze the executable.
Another aspect of insider threat is that the malicious agent will be aware of what analyses are routinely performed on the code base, and so will take pains to make sure the vulnerability is authored in such a way that it is undetectable by those analyses. Consequently there is a case for having a special “insider threat” analysis that is not available to all developers, but is instead run surreptitiously behind closed doors by highly trusted actors. Such an analysis would be specially configured to detect telltale signs of insider attack.

VI. CONCLUSION

Powerful static analysis for machine code is critical to supply chain risk management, both for software producers and for software consumers. Analyzing the exact software that will be executed provides the best fidelity in determining its strengths and weaknesses for the purpose of evaluation and comparison. Binary analysis allows organizations to devote their trust-validation efforts to supply-chain locations that are under organizational control. Furthermore, analysis at these locations can readily incorporate the contexts that are relevant to organizational needs. Performing analysis internally provides optimal levels of both security and control.

ACKNOWLEDGMENTS

CodeSonar for binaries is the culmination of many person-years of research and development carried out at GrammaTech and at the University of Wisconsin Madison, by too many individuals to list here. The authors wish to thank all who worked to make it a success.

REFERENCES