Preprint
UCRL-CONF-222171

Strengthening Software Authentication with the ROSE Software Suite

G.K. White

This article was submitted to
47th Annual Meeting of the Institute of Nuclear Materials Management, Nashville, Tennessee

June 2006

Strengthening Software Authentication with the ROSE Software Suite

Greg K. White
Lawrence Livermore National Laboratory
June 2006

Abstract

Many recent nonproliferation and arms control software projects include a software authentication regime. These include U.S. Government-sponsored projects both in the United States and in the Russian Federation (RF). This trend toward requiring software authentication is only accelerating. Demonstrating assurance that software performs as expected without hidden "backdoors" is crucial to a project's success. In this context, "authentication" is defined as determining that a software package performs only its intended purpose and performs said purpose correctly and reliably over the planned duration of an agreement. In addition to visual inspections by knowledgeable computer scientists, automated tools are needed to highlight suspicious code constructs, both to aid visual inspection and to guide program development. While many commercial tools are available for portions of the authentication task, they are proprietary and not extensible. An open-source, extensible tool can be customized to the unique needs of each project (projects can have both common and custom rules to detect flaws and security holes). Any such extensible tool has to be based on a complete language compiler. ROSE is precisely such a compiler infrastructure developed within the Department of Energy (DOE) and targeted at the optimization of scientific applications and user-defined libraries within large-scale applications (typically applications of a million lines of code). ROSE is a robust, source-to-source analysis and optimization infrastructure currently addressing large, million-line DOE applications in C and C++ (handling the full C, C99, C++ languages and with current collaborations to support Fortran90). We propose to extend ROSE to address a number of security-specific requirements, and apply it to software authentication for nonproliferation and arms control projects.

1. Introduction to Authentication

As we make progress toward the deployment of monitoring systems for nuclear material, two important goals must be observed: protection of the host country's sensitive information and assurance to the monitoring party that the nuclear material is what the host country has declared it to be. These goals are met by *certification* in the host country and *authentication* by the monitoring party. During both certification and authentication, each party needs to understand all of the operating parameters of all hardware and software in the deployed system. This paper concentrates on software authentication, but similar principles apply to hardware authentication, as well as to software and hardware certification.

Authentication is the process of gaining assurance that a system is performing robustly and precisely as intended. The simpler the system, the easier it is to authenticate. It is important to limit functionality to only what is needed to satisfy the requirements of the task. Each design decision makes authentication easier, or harder. For example, a design with Microsoft MS-DOS (which requires a 4.77-MHz processor and runs on a single 1.44-MB floppy disk) is significantly easier to authenticate than a Windows Vista (beta 2) installation (which requires an 800-MHz processor 512-MB of memory, and 15 GB of free disk space).[1] Simpler hardware, expressed in the number of gates, chips, or boards, is easier to authenticate than more complex hardware. The same can be said for software.

Other industries have a similar need for authentication. Computers that perform electronic voting[2] and gambling are disparate examples.

In my 2001 INMM papers,[3] I discussed a hypothetical perfect system for authentication, with transparent (to both parties) hardware and software development, and advocated "open source" hardware and software solutions.

In my 2005 INMM paper,[4] I advocated software language choices that lower authentication costs. I compared procedural languages with object-oriented languages. In particular, I examined the C and C++ languages, comparing language features, code generation, implementation details, and executable image size, and demonstrated how these attributes aid or hinder authentication. I showed that programs in lower level, procedural languages are more easily authenticated than object-oriented ones. I suggested some possible mitigations for using object-oriented programming languages.

2. What must be authenticated?

To authenticate a piece of application software most easily, both the source and binary versions are needed. Compilers and assemblers, the tools used to convert from the application's source to binary form are also needed. These must also be authenticated to a lesser extent, because they could also alter the code which will be executed. Other software which runs on the target system, such as the operating system and BIOS must be authenticated. To be complete, the compilers and assemblers used on the operating system and BIOS must also be authenticated. Often these compilers and assemblers are different from the one used for the application code.

In his classic paper[5] from 1984, Ken Thompson described that compilers are often written in the same language they compile. Binary versions of the compiler are often created using older versions of the same compiler. The GNU C compiler uses this approach. He shows that the compiler can be altered to look for a specific sequence of symbols in source code, and then alter the resulting binary. He specifically shows an example where the UNIX login program is "trojaned" by the compiler. (i.e., malicious code is embedded in the target binary.) He takes this attack one step further, creating a version of the compiler that trojans the login program, and also propagates the change into all future versions of the compiler, without altering their source code.

He concludes, "No amount of source-level verification or scrutiny will protect you from using untrusted code… I could have picked any program-handling program such as an assembler, a loader, or even hardware microcode. As the level of program gets lower, these defects will be harder and harder to detect." He then released a live version of the attack to a rival Bell Labs organization—an intramural prank that was never detected.[6]

In a recent paper[7], Dennis Wheeler revisits the Thompson paper, and offers a solution, *Diverse Double Compilation*, where the suspect compiler's source code is first compiled using a different, trusted compiler, then the suspect compiler source code is compiled using the compiler generated in the first step. If the two resulting binaries are bit-for-bit exact, then the compiler is no longer suspect. He suggests that security can be further enhanced by increasing the diversity of compiler implementation (for example, not sharing a common development heritage), development period (in time), the environment (operating system, processor and standard libraries), and carefully mutating the source code (whitespace, variable names, reordering statements) in ways that would not affect the resulting binary. He suggests that an overly simple compiler would be a good choice for the trusted compiler, because it contains less source code, and because performance is not important, in either the compilation process, or the resulting binary code.

If we do not resolve the inherited trojaned compiler problem by the method described in Wheeler, we are stuck with a chain of previous versions of the compiler which must also be authenticated.

3. How to authenticate software

Effective authentication requires that the developer *not* know the exact or complete specifics of the authentication process to be performed. The more the developer understands about the authentication measures, the more likely he is able to defeat the authentication. The authentication must not be done under an overly compressed or fixed timeline, nor can it be done solely in the presence of the developer. The parties to an agreement must allow and preserve a continuing ability to re-test the software into the future, should new concerns come to light, or new authentication tools or methods become available.

There are five primary methods of authentication of software:

- Extensive software testing with widely diverse inputs
- Visual inspection of the source code by a knowledgeable computer scientist
- Automated Code Coverage Tools
- Automated analysis of the source code
- Automated analysis of the resulting binary code.

Another software authentication method is technically feasible, but would never be used in a non-proliferation or arms control regime, because it involves

automated augmentation of the source code.[†] This is because augmentation of the source code would be done by the monitoring party (authentication). Thereafter the host country would no longer be able to trust the altered code in the production environment (compromised certification).

The first three methods are relatively well understood. In software testing, the software is subjected to a large number of diverse inputs to make sure the software performs correctly. Some inputs should be explicitly chosen by a knowledgeable domain expert in the application software to exploit known boundary conditions. Other input should be chosen randomly to further test the application software. Automation can simplify this kind of testing. The inputs might include some tests known to the developer ahead of time, but if all inputs were known, then a clever developer could exploit them. Even with random inputs, this kind of testing could not discover a backdoor. And even if it were possible to generate all inputs for testing, that would still be insufficient since it could be a combination of inputs that triggers a backdoor or bug. In 1985–1987, the Canada's Therac-25 radiation therapy machine was affected by this class of problem. The problem arose when the operator set an input level, and then cleared that value to put in another value. Because of a code flaw, a huge radiation dose was delivered instead. Three of these patients were believed to have died from the overdoses. This tragedy shows that, in effect, you have to try all combinations of possible inputs, a choice that grows as N! in the number of inputs.

Visual inspection by a knowledgeable computer scientist is also a useful method of authentication. The computer scientist must have a deep understanding of both application domain, and the implementation computer language. Moreover, obtusely written code, code with comments and variable names in a different natural language, or code with misleading comments, can make this task difficult to impossible even for an expert.

Automated code coverage tools evaluate test coverage, i.e., telling you whether there are segments of code that are not getting executed at all during a "normal operation" test suite.

Finally, automated analysis of the source and binary code complement each other as methods; some classes of exploits can be detected in either the source *or* the binary, while others require analysis of both. As another benefit, analyzing both source and binary code might lessen the need to authenticate compilers, assemblers, etc.

4. Automated analysis of the source code

Automated analysis of the source code by a software product will more easily find some classes of problems. These tools are needed to help this visual inspection process by highlighting code constructs that need additional and/or deeper scrutiny. The worst flaws and best concealed intentional trapdoors are likewise hidden by obfuscation techniques.[8] Some of these flaws and exploits can be avoided by putting restrictions on the language constructs used such as restricting the use of virtual functions in C++. This argues for applying a set of agreed coding standards to automated analysis, because the same tool used by the developer to enforce good programming practice could also be used during authentication to disclose suspicious code constructs. Expanding the rule set for authentication would provide still more assurance.

While some specific tests are unique to a specific regime, many tests are generic to any code inspection. Many tests also bring about the desired side effect of making the software more secure and robust.

4.1 Commercial tools for source code analysis

The development of software verification techniques has led to a number of commercial efforts to define tools that read source code and apply numerous proprietary tests.[9,10,11,12] These efforts to date have resulted in distinctly closed systems, protecting each company's intellectual property. The proprietary tools typically have limits on the order of 100K lines of code. This code size limit should not be a problem for arms control and nonproliferation regimes. However, many large software projects exceed this limit, so existing and derivative tools could not easily be applied to these codes.

4.2 DOE's ROSE software suite

Properly scaled for this challenge, ROSE[13] is a compiler infrastructure developed under DOE sponsorship, and originally targeted at the optimization

[†] An accepted design principle for detectors intended for use on classified objects is that the protection of the host country classified information is paramount. Thus, no authentication measure that would negate host country certification—such as alteration of detection source code—would be acceptable. [Ref: *The Functional Requirements and Design Basis for Information Barriers,* Pacific Northwest National Laboratory, May1999.]

of scientific applications and user-defined libraries within large-scale applications (typically applications of a million lines of code). ROSE is a robust, source-to-source analysis and optimization infrastructure currently addressing large, million-line DOE applications in C and C++ (handling the full C, C99, and C++ languages, with current collaborations to support Fortran90), and targeting a noncompiler audience. As a result, ROSE is extensible and uses a modular design to build custom optimization solutions for diverse applications. A Lawrence Livermore National Laboratory (LLNL) research project[14] will extend ROSE to address a number of security and authentication specific requirements and work with software analysis research groups to demonstrate its use on large-scale applications.

ROSE supplies a robust open infrastructure for source-to-source analysis and optimization, and thus could not only perform authentication and security analysis, but also automate transformations to make existing code more secure. Specific techniques include documenting specific security flaws for code reviews, instrumenting suspicious code for use in testing or production environments, and modeling applications using external verification tools (model checking, assertion testing, contract verification techniques, formal proof techniques, etc.). The automating of corrections to existing software could in many cases make it more secure (e.g., performing assertion testing on input buffers for buffer overflow, and switching standard unsecured library functions for more secure variants).

4.3 More examples of source code analysis

Source code contains information that is not in the binary. The binary treats memory as a linear array, whereas the source code expresses higher levels of granularity in structures (e.g., structs, class, unions). Recovering this nuanced information from the binary would likely be very difficult. Moreover, type information is largely lost in the binary, and the rules interpreting aliasing (because data of different types are not aliased) cannot be leveraged from it. Explicit or implicit casts that would compromise this are also lost in the binary (i.e., used by the compiler and thrown away).

One example of the need for source code analysis is the One-Definition Rule (ODR). The C and C++ language specifications stipulate that only one software module can define a template, type, function, or object. Unfortunately, no known compiler enforces this rule between two different files in the same program. Compilers are essentially defined to work on "well formed programs" as it can be expensive to detect all errors. Here, because detecting ODR violations requires whole program analysis, and because compilers typically operate on a single source file at a time, it is difficult to actually test for violations. If two software modules both include code to define the same structure, the one that is first in the linking process will succeed, and the other will be ignored. The order is specified not in the source code, but in the *Makefile*. This is not obvious to many programmers, and could easily evade a visual inspection. This is especially true where software modules are inspected individually. Only with whole program source code analysis, such as is available with ROSE, can this rule be properly tested.[15]

ROSE analysis can detect subtle exploitation of object-oriented language vulnerabilities. Dr. Quinlan's paper[16] describes a particularly simple compromise of the C++ class structure by violating ODR. The VPTR exploit replaces an object's virtual function table pointer (VPTR) with one containing malicious code.[17] The most trivial case redefines the existing definition of an inline virtual function. The paper shows a complete and working example of this exploit; it consists of 5 files, and under 60 lines of code. Existing C++ compilers do not catch this violation of the ODR rule, because they do not perform whole-program analysis. However, ROSE does. ROSE was the first product to identify the security flaw, and the first to enforce it in the C language.[18]

The most common exploit is the buffer overflow. In this attack, the code overwrites the end of a buffer, which can contain the return address of the function. By inserting a new address, the code can now call any other function. Tools could be written (using the ROSE infrastructure) that detect buffer overflows in source code, and could also detect standard library calls, which are susceptible to this kind of attack.

Another exploitation of particular concern to arms control and nonproliferation is the time-dependent control flow. An example of this would follow the rule: "Every Friday, pass the third container measured." This exploitation cannot be found through input testing, because it depends on the state of the clock. This kind of exploitation can only be found by tracing the effect of time-dependent variables through the control flow of a program. Again, source analysis tools are required to help measure the range of effect of time-dependent variables.

In his paper,[19] Lingxiao Jiang describes Osprey, a system for performing measurement unit checking on C source code. The programmer first attaches

measurement units to initial constants and variables. Osprey then automatically detects potential errors involving measurement units. They have shown good success in finding unknown measurement unit errors in mature computational physics codes. Since many arms control and nonproliferation regimes include a computational physics code, they could benefit from this analysis. In 1999, the Mars Climate Orbiter crashed into the surface of Mars, when one module of the software passed a variable using imperial units (feet, inches, etc.) to another module expecting the metric units.

5. Automated analysis of the binary code

There are a number of valid reasons to analyze binary code as well as source code. In some situations, only the binary version of the code is available (for instance, where proprietary commercial software is used in the system and the source code cannot be obtained). Another motivation is the need to ensure that the binary code exactly reflects the transformation of the source code. One example is where the compiler's optimizer eliminates writes to a data structure before freeing memory. In a famous example in Microsoft's Internet Explorer, a function stored an important password in heap memory. At the conclusion of the function, there was a source code segment specifically intended to overwrite the password on the heap with zeros. Unfortunately, the compiler's optimizer removed that segment during the optimization pass, leaving the password intact and vulnerable.

Some problems can *only* be detected in the binary version of the code. For example, binaries contain the binding of function names to locations in memory. Those bindings are vulnerable to subtle manipulations at runtime.

Another argument for analysis of the binary code is that C++ compilers have the freedom to determine when constructors and destructors should be called (though not the *order* in which they are called). This makes the source code a misleading indicator of the actual sequence of data manipulations, while the binary, though less transparent, is more reliable in this regard. Compiler-generated temporaries are a classic example of instances where the destructor may be called early. The compiler is also free to optimize away any redundant copy constructor calls. It seems likely that one could turn this to an advantage in an attack.

Each place in the language specification where the compiler has a degree of freedom represents an opportunity for an exploit, and there are quite a number of these in C and C++ (That said, C++ cleans up many C details—disallowing many—and has a very tight specification compared to the specification for Fortran90.)

Progress has been made in defining requirements and techniques for to analyzing binary files, but more work needs to be done. The IDA Pro[20] tool is useful, but it is unclear how to resurrect meaningful data from disassembled machine code. An LLNL-funded research project[21] will explore this topic in the coming year.

6. Other uses for authentication technologies

Authentication can also be used to improve cyber security in existing software applications. As operating systems become more robust, cyber attacks are increasingly targeting the more numerous applications that are correspondingly more vulnerable. Such applications are often not developed with the same attention to security as operating systems and are developed by smaller, more obscure companies lacking sufficient resources or expertise to address such complex security problems. Application software is also built on top of a huge software stack of user-written libraries, utility libraries not written by the software team, and system libraries. This compounds the security problem, because we are in essence trying to write secure software out of non-secure parts. Source and binary analysis are necessary to more fully understand the internals of these application software systems and to mitigate and find software flaws and intentional exploits.

7. References

[1] http://www.microsoft.com/windowsvista/getready/systemrequirements.mspx

[2] As an aside, a genius co-worker of mine stated, "If I wanted to rig an election with an electronic voting machine, and I could choose any computer language to write my hide my deception in, I'd do it in C++."

[3] White, G., Increasing Inspectability of Hardware and Software for Arms Control and Nonproliferation Regimes, *Proceedings of the INMM 2001 Annual Meeting*, Indian Wells, California

[4] White, G., Computer Language Choices in Arms Control and Nonproliferation Regimes, *Proceedings of the INMM 2005 Annual Meeting*, Phoenix, Arizona

[5] Thompson, K., Reflections on Trusting Trust, Turing Award Lecture, *Communications of the ACM*, Volume 27, Number 8, August 1984.

[6] http://en.wikipedia.org/wiki/Backdoor

[7] Wheeler, D., Countering Trusting Trust though Diverse Double-Compiling
[8] The Underhanded C contest http://bingweb.binghamton.edu/~scraver/underhanded/ and the Obfuscated V (vote) contest http://graphics.stanford.edu/~danielrh/vote/vote.html are two examples of contests to produce unexpected results from seemingly innocuous source code.
[9] http://coverity.com
[10] http://klocwork.com/
[11] http://www.polyspace.com/
[12] http://www.grammatech.com/products/codesonar/overview.html
[13] ROSE is not an acronym. http://www.llnl.gov/CASC/rose/
[14] Quinlan, et. al., An LDRD Proposal on Cyber Security for Software Security Analysis, June 2006
[15] Quinlan, et. al., Using Whole-Program Analysis to Detect Security Flaws, May 19, 2006.
[16] Quinlan, et. al., Using Whole-Program Analysis to Detect Security Flaws, May 19, 2006.
[17] Rix, Smashing C++ VPTRs, Phrack, May 2000.
[18] Quinlan, et. al., Software Security Analysis, LLNL LDRD Presentation, May 2006.
[19] Jiang and Su, Osprey: A Practical Type System for Validating Dimensional Unit Correctness of C Program, UC Davis, ISCE '06, May 2006.
[20] http://www.datarescue.com/idabase/
[21] Quinlan, et. al., An LDRD Proposal on Cyber Security for Software Security Analysis, June 2006

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.