LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-414623

Using Rose and Compass for Authentication

G. White

July 13, 2009

Institute of Nuclear Materials Management
Tucson, AZ, United States
July 13, 2009 through July 16, 2009

Using Rose and Compass for Authentication

Greg White
Lawrence Livermore National Laboratory
July 2009

Abstract

Many recent non-proliferation software projects include a software authentication component. In this context, "authentication" is defined as determining that a software package performs only its intended purpose and performs that purpose correctly and reliably over many years. In addition to visual inspection by knowledgeable computer scientists, automated tools are needed to highlight suspicious code constructs both to aid the visual inspection and to guide program development. While many commercial tools are available for portions of the authentication task, they are proprietary, and have limited extensibility. An open-source, extensible tool can be customized to the unique needs of each project. ROSE is an LLNL-developed robust source-to-source analysis and optimization infrastructure currently addressing large, million-line DOE applications in C, C++, and FORTRAN. It continues to be extended to support the automated analysis of binaries (x86, ARM, and PowerPC). We continue to extend ROSE to address a number of security-specific requirements and apply it to software authentication for non-proliferation projects. We will give an update on the status of our work.

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 side needs to understand all of the operating parameters of the 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 installation (which requires an 1 GHz 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 application and development software.

Other industries have a similar need for authentication. Computers that perform electronic voting[2] and gambling are disparate examples. In previous INMM papers,[3,4,5,6,7] we have discussed a hypothetical perfect system for authentication, with transparent (to both parties) hardware and software development, and advocated "open source" hardware and software solutions. We advocated software language choices that lower authentication costs, specifically comparing procedural languages with object-oriented languages. In particular, we 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. We showed that programs in lower level, procedural languages are more easily authenticated than object-oriented ones. We suggested some possible ways to mitigate the use of object-oriented programming languages. We described the scope of the software authentication process and the five methods of software authentication. We then concentrated on different types of source code analysis, introducing LLNL's ROSE software tool for automating the authentication of source code. Finally, we discussed how authentication of binaries is complementary to source code authentication.

2 - LLNL's ROSE software suite

Properly scaled for this challenge, ROSE[8] is a compiler infrastructure developed under DOE sponsorship, and originally targeted at the optimization of scientific applications and user-defined libraries within large-scale applications. ROSE is a robust, source-to-source analysis and optimization infrastructure currently addressing large, million-line DOE scientific applications in C, C++, and FORTRAN (PHP is also supported). ROSE is extensible and uses a modular design to build custom solutions for diverse applications. ROSE continues to be enhanced to address security and authentication-specific requirements. It has strong commercial and academic collaborations.

There are commercial tools which perform similar checks on source code. They are, for the most part closed systems which, at best, allow for minor customizations. None of these commercial tools support all of the languages needed for non-proliferation and arms control regimes (C, C++ and FORTRAN). Many of the tools are implemented on 32-bit Windows systems, which limit them to about 100K lines of source code. We feel that commercial tools complement analysis using ROSE, but are not a substitute for it.

ROSE supplies a robust open infrastructure for source-to-source analysis and optimization, and can perform authentication and security analysis. It can 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).

The main Intermediate Representation (IR) in ROSE is an Abstract Syntax Tree (AST) that preserves the detailed structure of the input source, including source file position and comment information. The AST's design enables source-based tool builders to accurately analyze and transform programs.[9] One of ROSE's key features is the ability to analyze an entire program's source code, by merging the ASTs of each of the many individual files which make up the source code. This has the additional benefit of significantly decreasing the memory usage, and simplifying analysis of the resulting AST. On one large application, the merged AST was almost 3 times smaller than the sum of the individual file's ASTs.[10]

Another program analysis result is the Control Flow Graph (CFG). The CFG represents all paths that might be traversed through a program during its execution.[11] The System Dependence Graph (SDG) can be used by ROSE to perform program slicing. A program slice determines either all source code that might affect a given variable at a particular point in the execution ("backward slicing") or all variables that could be affected by a given variable at a particular point in the source code ("forward slicing").[12] Program slicing can help the computer scientist better understand how a program works and what it does, since it allows the computer scientist to break the code into smaller, easy to understand subsets.[13,14] This aids the computer scientist in authentication by a greater understanding of the code through visual inspection. Many legacy computer programs include lots of source code statements which were once used, but are no longer contribute to the results of the program. Without automation, finding this particular kind of dead code is tedious and time consuming. Backward slicing is one method of automating this process. Removing dead code creates a simpler program which is easier to authenticate, consumes less memory, and runs faster.

ROSE uses two other kinds of graphs to perform software analysis. A Control Dependency Graph is used to determine if Statement B's execution is dependent on Statement A. This is shown in this simple example:

```
A:   IF (x>0)
B:       printf("x is positive");
```

A Data Dependency Graph shows what variables are dependent on the values of other variables. For instance in the standard equation for solving a quadratic equation ($ax^2+bx+c=0$), we use the equation:

[†] An accepted design principle for programs intended for use on classified objects is that the protection of the host country's classified information is paramount. Thus, no authentication measure that would negate host country certification—such as alteration of source code—would be acceptable. [Ref: *The Functional Requirements and Design Basis for Information Barriers,* Pacific Northwest National Laboratory, May 1999.]

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In the Data Dependency Graph, the variable *x* is data dependent on the variables *a*, *b*, and *c*.

ROSE continues to improve the robustness for its support for binary analysis. Analysis of large (4.5M instruction) binaries has been demonstrated. ROSE includes support for Intel *x86*, ARM, and PowerPC processors. This covers the vast majority of processors used from embedded processors (cell phones, PDAs, automobiles, etc.) to supercomputers. ROSE supports both Windows PE and ELF binary file formats. These two binary file formats cover a majority of operating systems. Just like source code analysis and transformations, ROSE can perform binary analysis. The ROSE team has written about ten Compass checkers to analyze binaries. They have created a robust infrastructure to analyze binaries. Writing Compass checkers for binaries is more difficult to write than Compass checkers for source code. In fact, ROSE can support combined analysis on both the source and binary versions of a program. Work by the ROSE team this year has included a comprehensive comparison of separate compass checkers (both source code and binary checkers) for the same bad programming practices[15]. This comparison found most bad programming practices were found by both the source code and the binary checkers. Some were found by only one of the checkers (either binary or source code).

Before this year, ROSE only supported static analysis of source code and binaries. Dynamic analysis was handled by augmenting the source code, and then analysis was performed after execution. The ROSE team has begun work on dynamic (i.e. runtime) analysis of binaries, and combined static and dynamic analysis. This will provide a powerful analysis capability to problems of interest.

ROSE continues to enhance its support for multiprocessing. This allows ROSE to produce faster results and work on larger and larger codes, especially for analysis which requires a large memory footprint. Compass takes advantage of cache coherency to run multiple checkers at once, obtaining up to a 25*x* speedup.

Work on the Compass subproject continues. This project will create a collection of tests for detecting bad programming practices in source code. Many of these bad programming practices are defined in the SAMATE Reference Dataset Project, hosted by NIST[16], and the Secure Coding Standards for C and C++ at CERT[17]. There are thousands of examples of bad programming practices. LLNL, NIST, and CERT are collaborating on software security. As of the writing of this paper, there are over 150 Compass Checkers. The ROSE team welcomes external contributions to this sub-project.

3 - A Simple Compass Checker

One of CERT's Secure Coding Rules for C[18] and C++[19] is that every *switch* statement should have a *default* clause unless every enumeration value is tested. Here are examples of complaint and non-complaint code:

Compliant Code	**Non-compliant Code**

`switch (language) {`
`  case LANG_ENGLISH:`
`    printf("No"); break;`
`  case LANG_RUSSIAN:`
`    printf("Nyet"); break;`
`  default:`
`    printf("???"); break;`
`}`
	`switch (language) {`
	`  case LANG_ENGLISH:`
	`    printf("No"); break;`
	`  case LANG_RUSSIAN:`
	`    printf("Nyet"); break;`
	`}`

The relevant code in a Compass Checker would look like this:

```
class visitorTraversal : public AstSimpleProcessing {
public:
   virtual void visit (SgNode* n) {
      SgSwitchStatement* s = isSgSwitchStatement (n);
      if (s) {
         SgStatementPtrList& cases = s->get_body ()->get_statements ();
         bool switch_has_default = false;
```

```
        // 'default' could be at any position in the list of cases.
        SgStatementPtrList::iterator i = cases.begin();
        while (i != cases.end () && !switch_has_default) {
            if (isSgDefaultOptionStmt (*i))
               switch_has_default = true;
            ++i;
        }
        if (!switch_has_default)
           // Report non-compliant code
           output->addOutput(new CheckerOutput(n));
        }
     }
  }
```

This checker locates *switch* statements that do not contain a *default* clause, but does not check that every enumeration is tested. This would be a more complicated checker.

4 - Experiences with ROSE and Compass

We continue to increase our knowledge of ROSE and understand its inner workings. ROSE targets computer scientists with a competent object-oriented programming background, but not necessarily an expert background in compiler theory. Each checker tends to expose a new part of ROSE's complex object hierarchy. After writing about a dozen checkers, computer scientists start to understand the "big picture". We have found that it is often easy to write naïve, incomplete checkers. Writing complete, comprehensive checkers requires a deep understanding of the problem, and good peer review.

In our 2005 INMM paper,[20] we listed a number of recommendations for programmers who are writing in C and C++. We have listed them below for reference, along with their current status. These checkers are being used by other computer scientists at LLNL as part of projects that are similar to authentication.

C Language	
Requirement	Implementation/Status
Encourage the use of system calls which do not allow for buffer overflows (*gets* vs *fgets*, *strcpy* vs *strncpy*, etc.).	Enhanced an existing Compass checker.
Turn off compiler optimizations.	Manual inspection of the build system
Use only static loading, no dynamic loading of object files or Dynamic Loaded Libraries (DLL).	Enhanced an existing Compass checker to detect dynamic library loading at runtime. Need to write a binary Compass checker to check for dynamic loading in the linking phase.
Use a *malloc*() library that detects buffer underflow and overflow [e,.g. *malloc_debug*].	Enhanced an existing Compass checker.

C++ Language	
Requirement	Implementation/Status
Don't use virtual methods.[21]	Wrote a new Compass checker
Restrict the use of overloading of functions to help reduce name confusion	Implemented as a more comprehensive checker to detect overloaded functions
Don't use default arguments in functions	Wrote a new Compass checker
Do not use overloaded operator *new()* except in system and STL headers	Implemented as a more comprehensive checker to detect overloaded operators

C Language	
Consider self-check of dynamic executable's MD5/SHA checksum.	Not applicable to static source or binary analysis.
Dynamic casting of C pointers should be discouraged.	Requires an enhancement to ROSE
Encourage the liberal use of assertions[22] [e.g. design-by-contract] to verify that pointers are non-null, type values are consistent, etc.	Requires an enhancement to ROSE
Be cautious with the use of asynchronous signal handlers and the "volatile" data type designation.	Wrote two new Compass checkers to detect this rule.

C++ Language	
Do not allow dynamic casting of pointers in C++	Requires an enhancement to ROSE
Discourage the use of templatization outside the STL.	Wrote a new Compass checker

5 - Next Steps for using ROSE for Authentication

We have identified additional properties which can be enforced on C and C++ source code to add additional reliability and security. If enforced, they will provide additional assurance that the code performs as declared. This task will include the creation and adaptation of software tools which are extensions to the LLNL ROSE software package that will help knowledgeable computer scientists perform software authentication in support of non-proliferation regimes. It will both automate many of the manual checks of the software and highlight specific sections of software which need additional scrutiny. Examples of these properties are:

- Restrict casts on pointers (with a special case for NULL)
- Restrict function pointers
- White list of accepted function calls and global variables
- Enforced constraints on specific functions
- Detect One-time Definition Rule (ODR) violations
- Unique names for variables (to avoid opportunities for confusion)
- No macros of keywords
- Enforce/restrict macros in inappropriate locations in source code
- Ill-formed macros
- Static sizes for arrays
- Integer overflow detection (from CERT's Secure Coding in C).

The manual process of visual inspection of the code can be made more productive by creating tools which aid the knowledgeable computer scientist in understanding the inner workings of the code. During the authentication of the source code, the knowledgeable computer scientist will run the code many times with widely varied inputs to understand how the program behaves. In addition to running the unmodified code as it was delivered by the developers, the computer scientist may benefit by running the code with automatically generated enhancements to the source code, which may enforce array bounds at runtime. This will even increase confidence in the unmodified code, since the code has already demonstrated that it behaves appropriately within the given test cases. We will use ROSE to automatically create these kinds of tools and transformations.

ROSE can be customized to create tools which enhance the manual/visual inspection of source code by using forward and reverse slicing. Slicing allows a computer scientist to pick a location in the source code, and a variable and ask the questions: "What source code had influence in the value of this variable at this point in the execution?" (reverse slicing), and "What variables will be affected by this variable as the program continues to execute" (forward slicing). These checkers will utilize the Dataflow Graphs implemented in ROSE to perform this task.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] http://www.microsoft.com/windows/products/windowsvista/editions/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 if I could choose any computer language to write in to 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] White, G., Strengthening Software Authentication with the ROSE Software Suite, *Proceedings of the INMM 2006 Annual Meeting*, Nashville, Tennessee

[6] White, G., Tools and Methods for Increasing Trust in Software, *Proceedings of the INMM 2007 Annual Meeting*, Tuscon, Arizona

[7] White, G., Tools for Authentication, *Proceedings of the INMM 2008 Annual Meeting*, Nashville, Tennessee

[8] ROSE is not an acronym. http://www.rosecompiler.org

[9] Quinlan, D., Vuduc, R., Techniques for Specifying Bug Patterns, *PADTAT'07*, London, England, July 2007

[10] Panas, T., Quinlan, D., and Vuduc, R., Analyzing and Visualizing Whole Program Architectures, *International Workshop on Aerospace Software Engineering*, May 2007

[11] http://en.wikipedia.org/wiki/Control_flow_graph

[12] http://www.grammatech.com/research/slicing/slicingWhitepaper.html

[13] Horwitz, S., Reps, T., and Binkley, D., Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems* 12, 1 (January 1990), 26-60.

[14] Binkley, D., Gallagher, K., Program Slicing, http://www2.umassd.edu/SWPI/slicing/loyola/survey.pdf

[15] Thomas Panas and Dan Quinlan. Techniques for Binary Software Quality Analysis: Windows and Linux. *Workshop on Defects*, Chicago, 2009

[16] http://samate.nist.gov/SRD/

[17] https://www.securecoding.cert.org/confluence/display/seccode/CERT+Secure+Coding+Standards

[18] https://www.securecoding.cert.org/confluence/display/seccode/MSC01-A.+Strive+for+logical+completeness

[19] https://www.securecoding.cert.org/confluence/display/cplusplus/EXP08-A.+A+switch+statement+should+have+a+default+clause+unless+every+enumeration+value+is+tested

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

[21] Although this may seem to preclude all inheritance since C++ only allows inheritance of classes with virtual methods, a small exception can allow one "useless" method of the form "virtual void useless(void) {}" to allow inheritance where required.

[22] In non-proliferation and arms control regimes, one could argue that getting an error condition is significantly better than getting the wrong answer.