LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-405315

Tools for Authentication

G. White

July 11, 2008

Institute of Nuclear Materials Management
Nashville, TN, United States
July 13, 2008 through July 17, 2008

Disclaimer

Tools for Authentication

Greg White
Lawrence Livermore National Laboratory
July 2008

Abstract

Many recent Non-proliferation and Arms Control 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 (projects can have both common and custom rules to detect flaws and security holes). Any such extensible tool must be based on a complete language compiler infrastructure, that is, one that can parse and digest the full language through its standard grammar. ROSE is precisely such a compiler infrastructure developed within DOE. ROSE is a robust source-to-source analysis and optimization infrastructure currently addressing large, million-line DOE applications in C, C++, and FORTRAN. This year, it has been extended to support the automated analysis of binaries. We continue to extend ROSE to address a number of security-specific requirements and apply it to software authentication for Non-proliferation and Arms Control 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] 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

LLNL-CONF-405315

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[7] 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 and C++. New for this year is full support for PHP and FORTRAN. It targets computer scientists with a competent object-oriented programming background, but not necessarily an expert background in compiler theory. ROSE is extensible and uses a modular design to build custom solutions for diverse applications. A Lawrence Livermore National Laboratory (LLNL) research project[8] extends ROSE to address a number of security and authentication-specific requirements will include collaborations 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 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 continues to improve its support for binary analysis. It includes support for both Intel *x86* and ARM processors, and 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. In fact, ROSE can support combined analysis on both the source and binary versions of a program.

ROSE now supports both symmetric multiprocessing (where all processor cores share common memory) and distributed multiprocessing (where processor cores communicate over a high-speed network) tested to 256 processors. This allows ROSE to produce faster results and work on larger and larger codes, especially codes which

[†] 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.]

are larger than the available memory in a single computer. Compass takes advantage of cache coherency to run multiple checkers at once, obtaining to 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[15], and the Secure Coding Standards for C and C++ at CERT[16]. 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 about 125 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[17] and C++[18] 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 started on the learning curve with ROSE by writing a Compass checker which implemented one of the coding guidelines for a large scientific code at LLNL. The rule stated that "*for()* construction loops must only include statements that control the loop. In particular, *for()* loops must not initialize or increment/decrement variables not directly related to the loop control." Here are two examples of code segments which do and do not comply with this rule.

Compliant Code	Non-compliant Code

`s=0;`
`for (i=0; i<100; i++) {`
`  a[i]=100-i;`
`  s=s+a[i];`
`}`
	`s=0;`
	`for (i=0; i<100; s=s+a[i], i++) {`
	`  a[i]=100-i;`
	`}`

This provides a good example of how rules are redefined when they are converted for static source code analysis. A naïve approach might be to notice that the second example has the comma operator, so any code with a comma operator in a *for()* statement is non-compliant. Instead, we implemented a Compass checker which assumed that variables that were assigned in the initialization phase of the *for()* statement were related to the loop control. We collected these variables in a set. We then collected the variables that were altered in the increment/decrement phase of the *for()* statement. An error was generated listing any variables in the second set that were not in the first set. We believe this approach enforces the spirit of the coding standard, but additional restrictions could be enforced by the Compass checker. This also shows how Compass checkers can be enhanced over their lifetime.

The *for()* statement checker gave us an excellent introduction to writing checkers. We started with a simple traversal checker (one that visited each node only once), but quickly came to the conclusion that we needed a nested traversal, where the abstract syntax tree for each phase of the *for()* statement is separately tree-walked and then the combined result is given. It also gave us an introduction to the collection of possible AST nodes in a *for()* statement. With this success, we found a new target for checkers.

In our 2005 INMM paper,[19] 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:

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.

C++ Language	
Requirement	Implementation/Status
Don't use virtual methods.[20]	Wrote a new Compass checker
Restrict the use of overloading of functions to help reduce name confusion	TBD, will be implemented as a more comprehensive checker to detect overloaded functions
Don't use default arguments in functions	TBD

C Language	
Requirement	Implementation/Status
Use a *malloc*() library that detects buffer underflow and overflow [e,.g. *malloc_debug*].	Enhanced an existing Compass checker.
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, should be implemented in the coming year.
Encourage the liberal use of assertions[21] [e.g. design-by-contract] to verify that pointers are non-null, type values are consistent, etc.	Requires an enhancement to ROSE, should be implemented in the coming year.
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	
Requirement	Implementation/Status
Do not use overloaded operator *new()* except in system and STL headers	TBD, will be implemented as a more comprehensive checker to detect overloaded operators
Do not allow dynamic casting of pointers in C++	Requires an enhancement to ROSE, should be implemented in the coming year.
Discourage the use of templatization outside the STL.	TBD

One of the checkers was a simple traversal of the AST, while others required a different kind of traversal. In this kind of traversal, attributes are inherited from a node's parents. Each checker we write exposes more of the ROSE software, and teaches more complex techniques for writing checkers. A recent enhancement to ROSE allows Compass checkers to differentiate types, functions, and data structures by their origin (i.e. application, libraries, compiler, operating system, etc.).

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] ROSE is not an acronym. http://www.rosecompiler.org
[8] Quinlan, et. al., An LDRD Proposal on Cyber Security for Software Security Analysis, June 2006
[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] http://samate.nist.gov/SRD/
[16] https://www.securecoding.cert.org/confluence/display/seccode/CERT+Secure+Coding+Standards
[17] https://www.securecoding.cert.org/confluence/display/seccode/MSC01-A.+Strive+for+logical+completeness
[18] https://www.securecoding.cert.org/confluence/display/cplusplus/EXP08-A.+A+switch+statement+should+have+a+default+clause+unless+every+enumeration+value+is+tested
[19] White, G., Computer Language Choices in Arms Control and Nonproliferation Regimes, *Proceedings of the INMM 2005 Annual Meeting*, Phoenix, Arizona
[20] 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.
[21] In non-proliferation and arms control regimes, one could argue that getting an error condition is significantly better than getting the wrong answer.