SAMATE's Contribution to Information Assurance

by Paul E. Black

There is far too much software in today's information world to check manually. Even if people had the time to inspect thousands or millions of lines of code, nobody could remember all the constraints, requirements, and imperatives to make sure the software is secure. Automated tools are a must.

These tools can help design and build the right software in the first place, for instance, checking protocols, consistency with rules, and properties. Preventing flaws at the beginning of the software life cycle is the best way to get high quality and highly reliable software.

But what if the system being designed includes commercial, off-the-shelf (COTS) packages? How can a contractor thoroughly audit or check large packages from subcontractors? What kinds of flaws does the current development process leave? Does a new software process yield better quality software? To address these questions, the finished software must be checked. Again, the quantity of software requires automated software checking or at worst manual checking of exceptional instances found by automated means.

To be sure, testing is a vital part of assurance, too. If one does not have access to the source code, which is often the case with COTS packages or Web services, testing may be the only feasible way to gain assurance. Even when the source code or the binary are available, testing can be closer to actual use. Testing can catch configuration or system problems that are taken for granted when code is examined. On the other hand, reviews can find problems that are unlikely to be found by testing. For instance, a malicious backdoor that grants special access for a particular user name, say "matahari," cannot feasibly be found by functional, or black box, testing.

Another advantage of automated tools is that they can be updated and rerun relatively quickly when a new type of flaw is discovered or the security policy is changed. It is impractical to recheck everything manually for apparently minor changes in the system.

The SAMATE Project

Which tools find what flaws? Backing up, what is the list of all flaws to be found? Can tools check compliance with internally developed style or guidelines? If a tool passes a system with no outstanding alarms, how secure is system, really? Is the new version of a tool "better" than the preceding version?

The National Institute of Standards and Technology (NIST) Software Assurance Metrics and Tool Evaluation (SAMATE) project seeks to help answer these and other questions. The SAMATE Web site [1] explains that the project, begun in late 2004, is largely funded by the Department of Homeland Security (DHS) to help identify, enhance and develop software security assurance (SSA) tools. NIST is leading in (A) testing software evaluation tools, (B) measuring the effectiveness of tools, and (C) identifying gaps in tools and methods.

Although much work has been done in these areas, there is little coordinated, comprehensive, thorough, and objective work uniting all these. Instead we see isolated papers comparing different tools, surveys of methods and techniques, endorsements and experience reports, and best practices Web sites.

The SAMATE project is producing and catalyzing:

- a common enumeration of software weaknesses and flaws
- a taxonomy of SSA tools
- a survey of SSA tools and companies
- specifications of SSA tool classes
- detailed test plans and test sets for SSA tool classes
- metrics and measures for software and for SSA tools
- white papers pinpointing gaps in tool functions and proposing research requirements for new tools and new tool classes
- proposals for experiments and studies

Workshops

This project's scope is very broad, and our particular group in NIST does not have as much background in security and software assurance as others. To build collaborations and reach community

consensus, SAMATE has held several public workshops.

The first workshop, in August 2005, examined the state of the art in security assurance tools, particularly those that detect security flaws and vulnerabilities. The workshop was also the beginning of a standard reference dataset of programs with known flaws. Forty-five people attended, including representatives from the federal government, universities, more than a dozen tool vendors and service providers, and many research companies. The proceedings, including presentations and meetings notes, are published as NIST Special Publication 500-264. [2]

In October, we sponsored and hosted an Open Web Application Security Project (OWASP) conference.

In Long Beach in November 2005, we produced a workshop co-located with the Institute of Electrical and Electronics Engineers (IEEE) Automated Software Engineering (ASE) conference. This workshop convened researchers, developers, and government and industrial users of software security assurance (SSA) tools to discuss and refine the taxonomy of flaws and functions, come to a consensus on which SSA functions should first have specifications and standard tests developed, gather source code analyzer tool developers for "target practice," see how reference datasets fare against various tools, and identify gaps or requirements for research funding.

Working with others, we brought a very early version of the software reference dataset (SRD). Participants ran their tools against a subset of the SRD to demonstrate the state of the art in finding flaws and to provide suggestions for extensions to and improvements of the SRD. [3]

We held a Static Analysis Summit on 29 June 2006 in Gaithersburg, Maryland. [4]

A Taxonomy of Flaws

To accurately determine how well a tool checks for flaws, one must begin with a taxonomy of flaws. A taxonomy is not merely a list, but an organization into classes with shared characteristics. For instance, buffer overflow is a well-known (and unfortunately still widely occurring) type of flaw. But the classification "buffer overflow" can be further refined into heap or stack overflows, underflows or overflows, *etc.* In fact, the *CLASP Reference Guide* [5] lists eight different types of overflows. Even finer distinctions may be important to language designers or tool researchers, but may be unimportant to the programmer.

Authors have created and published many taxonomies of flaws. [6] [7] [8] For instance, MITRE grouped repeated problems listed in the Common Vulnerability and Exposures (CVE) [9] into a list of vulnerability examples. These works approach the problem from different views and define flaws differently, but have limited effort to reconcile the definitions, classifications, and details. SAMATE workshops catalyzed work to come up with one common enumeration of weaknesses. [10] Over time the taxonomy is sure to expand and change, but work can be shared instead of starting over for each good idea.

A Taxonomy of SSA Tools

Having a taxonomy of weaknesses, can we start testing tools? In a project of such ambitious scope, effort must be prioritized: we must choose which kinds of tools to look at first and which must be left for the future. Then, how do we choose which classes to work on? We must be able to list all classes of SSA tools so we can rationally (or at least, coherently) decide which ones *not* to work on. It follows we must also have a taxonomy of software assurance tools. The proposed taxonomy is organized around four facets:

- life cycle phase
- automation level
- approach
- viewpoint

The life cycle phase corresponds to the type of artifacts used, *e.g.* specifications, source code, executable, *etc.* It is documented as a simple waterfall model, even though more elaborate models are often better for the software process.

The automation level is a simple classification of how much human expertise,

effort, or knowledge is required. Level 0 is manual procedures, like code review. Sometimes there is no replacement for human involvement. The next levels have varying degrees of automation:

1. analysis aid
2. semi-automated
3. automated

Level 1 is analysis aids that help human analysts, but have no particular software assurance function themselves. Some examples are call-graph extractors, configuration control systems, or random test generators. Semi-automated tools or techniques at level 2 are targeted toward assurance, but need varying degrees of human judgment for extreme cases or to make a final decision. Most code and Web scanners fall in this category. They may point out things that are certainly flaws, but in other cases can issue only warnings about potential flaws. A human must then check and make the final determination. Finally, a firewall is an example of a completely automated tool at level 3. It takes action on whether to pass, trash, or cache packets without human intervention. Manual setup or auditing of automated tools does not make them semi-automated.

A tool may take four different approaches to software assurance: preclude the flaw from possibly occurring, detect a flaw or its exploit and report it, mitigate flaws to reduce or eliminate damage, and react to a flaw or its exploit. Choosing another language instead of C precludes most buffer overflows. Source code and Web scanners take the approach of detecting flaws. A multi-level security system can mitigate many security flaws. Finally, an intrusion-detection system reacts to exploited flaws by denying access.

The final facet, viewpoint, is either internal or external. An external viewpoint corresponds to functional or "black box" testing or Web penetration testing. Code reviews and intrusion detectors are prime examples of tools that work from an internal viewpoint.

Testing an SSA Tool Class

With a coherent taxonomy of software security assurance tools, we can rationally decide which classes of tools are most important, which to do first, and which to leave for later.

When we have chosen a particular class of tools to work on, we begin by writing a specification. The specification typically consists of an informal list of features, for quick orientation, then more formally worded requirements for features, both mandatory and optional. Specifications often include a glossary and a section with technical background, which gives a tutorial introduction.

For each tool class, we also recruit a focus group to review and advise on specifications. Group members are developers, academic researchers, and users. We depend on their expertise to make sure the specifications are widely acceptable.

While we are developing a specification, we also work on a test plan and test sets. What do current commercial and research tools of this class do? How will we test this kind of tool? This practical work helps us understand the specification. Once the focus groups review the specification and we incorporate public comment, we develop a test plan. A test plan details how a tool or technique is tested, how to interpret test results, and how to summarize or report tests. Most test plans require a test suite, which is a set of test cases. For example, code analyzers require a test suite of dozens or hundreds of large and small examples of source code with known flaws. The test suite also includes examples that are free of flaws to test for false alarms. Web penetration testers need executable applications with known flaws, like WebGoat. [11]

Currently we are developing a specification and test plan for source code analyzers. The first draft should be available at the Static Analysis Summit. [4] We are also developing a specification for Web application scanners.

A Standard Reference Dataset

While developing suites of tests, we collect much larger numbers of candidate test cases. This collection, the SAMATE Reference Dataset (SRD) [12], is freely accessible online. So far, we have collected more than 1,400 test cases, which academic researchers, tool developers, and tool evaluators can freely access to develop new methods and compare results. New test cases are constantly being added. The SRD allows anyone to search the test cases on a number of criterion, select any combination, and download them. Upon approval, researchers will be given accounts to contribute to the SRD. The SRD is a repository and clearing house for samples of designs, code, binaries, and other artifacts to accelerate research and development.

A single test case can have explanatory information associated with it, for instance, the author or contributor, the date submitted, language, which flaw(s) it exhibits, and a description. In addition, test cases may have directions on how to compile and link source code, input that triggers the flaw, or expected output. Users also will be able to add comments on a test case.

For historical stability, the content of test cases will never be updated. If the code in a test case needs to be fixed or improved, a new test case will be added, and the status of the existing test case will be changed to "deprecated." Deprecated status advises against using the case for any new work. A reference to the new test case will be added to the deprecated case. This way, a test report referring to a certain test suite can be rerun exactly, even years later. Although the metadata may be changed or comments added, the original test case won't be changed.

Future Challenges

Ultimately, these tests for classes of tools and techniques exist to help answer real questions. Is a program secure (enough)? How secure does technique *X* make a program? How much more secure does technique *X* make a

program after doing *Y* and *Z*? How much assurance does tool *T* give? Dollar for dollar, can I get more reliability from methodology *P* or methodology *S*?

We will work with others on developing and validating metrics and measures, not only for software and designs, but also for the tools themselves. Possible measurable qualities for tools and techniques are effectiveness (do they find important flaws), completeness (how many kinds of flaws can they find? Do they catch all of those kinds?), soundness (ratio of false alarms to real weaknesses found), report precision (location, severity, and type of flaw), and scalability and maximum size of artifact that can be handled. We would also like to characterize the ability of the user to trade completeness for soundness, add their own rules and style policies, and set a severity cut-off points.

Throughout our investigation, we will find gaps and opportunities in tools and techniques. We will write papers detailing these gaps and research opportunities. We will also propose requirements for research funding to develop new tools, do studies or experiments, or explore methods for assuring information. With more than a century of experience in measurement science and standards, NIST is uniquely qualified to conduct or collaborate in studies and experiments to improve the foundation of computer science and software assurance. ■

References

1. "SAMATE," *http://samate.nist.gov/* accessed 29 March 2006.
2. *Proceedings of Defining the State of the Art in Software Security Tools Workshop*, Paul E. Black (chair) and Elizabeth Fong (ed), NIST Special Publication 500-264, November 2005. Available at *http://hissa.nist.gov/~black/Papers/nistSP500-264_aug05.html*
3. *Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*, Paul E. Black (chair), Michael Kass (co-chair), and Elizabeth Fong (ed), NIST Special Publication 500-265, February 2006. Available at *http://hissa.nist.gov/~black/Papers/nistSP500-265_nov05.html*
4. "Static Analysis Summit," *http://samate.nist.gov/SAS* accessed 29 March 2006.
5. John Viega, *CLASP Reference Guide: Volume 1.1 Training Manual, Secure Software*, McLean, VA, 2005.
6. Huaiqing Wang and Chen Wang, *Taxonomy of Security Considerations and Software Quality, Communications of the ACM*, 46(6):75-78, June 2003.
7. Katrina Tsipenyuk, Brian Chess, and Gary McGraw, *"Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors," Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics, op. cit.*, pp 36-43.
8. Michael Howard, David LeBlanc, and John Viega, *19 Deadly Sins of Software Security.* McGraw-Hill Osborne Media, July 2005.
9. "Common Weakness Enumeration," *http://cve.mitre.org/cwe/* accessed 31 March 2006.
10. "WebGoat Project," *http://www.owasp.org/software/webgoat.html* accessed 29 March 2006.
11. "SAMATE Reference Dataset," *http://samate.nist.gov/SRD/* accessed 29 March 2006.

About the Author

Dr. Paul E. Black | is a computer scientist at National Institute of Standards and Technology (NIST). Before joining NIST, he had nearly 20 years of industrial experience in software for integrated circuit design and verification, quality assurance, and business data processing. He earned a PhD from Brigham Young University in 1998. He has published in software testing, formal methods, software verification, and quantum computing, and has taught at Johns Hopkins University. He is a member of Association of Computing Machinery (ACM), the Institute of Electrical & Electronics Engineers (IEEE), and IEEE Computer Society.