# SOFTWARE ASSURANCE WITH SAMATE REFERENCE DATASET, TOOL STANDARDS, AND STUDIES

*Paul E. Black, National Institute of Standards and Technology (NIST), Gaithersburg, MD*

## Abstract

Today's avionics systems depend more and more on software from many sources: vendors, subcontractors, in-house, and open source. System interactions are exposed to external agents in contexts from air-to-ground links to OS patches downloaded via the Internet. This is a huge amount of software with the risk of attack from distant global sites. Yet users need assurance that the software will work and not create security problems.

We focus on NIST's Software Assurance Metrics And Tool Evaluation (SAMATE) project and its contribution. SAMATE is developing specifications, metrics, and automated test suites for software assurance tools. For instance, source code security analyzers can help developers produce software with fewer security flaws. They can also help identify malicious code and poor coding practices that lead to vulnerabilities. The project's publicly available reference dataset, the SRD, contains more than 1800 flawed (and fixed!) program examples to help evaluate software assurance tools and algorithms. These metrics and reference datasets help purchasers confirm tool vendors' claims. We also study the assurance impact of tool use, methods, and techniques.

## How Can One Get Good Software?

The qualities needed in today’s software and systems cannot be “tested in”. Desirable properties, such as security, safety, and reliability, must be designed in and built in from the beginning. The development process must be such that users can rely upon the resulting systems or software.

Is the user then left to depend on blind faith in artifacts produced? Clearly, no. Even with the most disciplined and well-characterized processes, artifacts must be examined to gain assurance that the output of the process is close to the target qualities. This is the essence of quality control.

Concepts from quality control apply to software, although for different reasons. Software is not subject to manufacturing variations in the same sense as bullets or pencils. However today’s software and systems are far too complex to test absolutely every possibility or to formally verify everything. Discipline in the development process must compensate for post-production limits to testing and verification. Such limits are not unique to computers. Post-production assurance of hand grenade quality is also limited to testing only a sample of the production.

### *Role of Post-Production Analysis*

Software must be tested, reviewed, verified, and otherwise analyzed after production to assure that desired levels of quality, safety, security, etc. are met, that no changes in the process or the environment have lowered the quality of the output. If we find assurance of required properties, the artifacts may be released. However if the post-production analysis does not provide the assurance we need, some remediation may be feasible. In some cases, it may be cheaper and faster to reject the product, look for and address root causes, and begin again.

Such analysis builds a strong assurance case quickly when the development process is well known and well characterized. But what can be done if the software is commercially acquired? Even a contractor needs to be qualified. What if there is a new software process to qualify? How can we gain assurance that legacy systems do and will perform acceptably in today’s complex systems and in the environment of electronic aggression from distant corners of the globe? In all these cases, and many more, we must rely largely on examination of the primary artifacts: the source code or binaries supplied. Examination may be some combination of testing, static analysis, review, and formal reasoning. Each of these approaches have different techniques which range from completely manual through machine assisted to highly or completely automated. For these reasons, NIST’s Software

Assurance Metrics and Tool Evaluation (SAMATE) project began by characterizing analysis, especially tools, applicable after production [1].

### *Assurance in Hostile vs. Benign Environments*

Traditional software development is implicitly for a benign environment. The biggest threat is accidentally triggering a latent error. The assumption is that Nature does not *try* to cause system failures. Although collective wisdom in such forms as Murphy's Law cautions us against complacency, the world today is much worse. The wide spread decay of morals and social values of honesty, vigilance, and restraint allow serious behavior to flourish and the Internet amplifies its global effect. No longer is the threat merely adolescents working for bragging rights or seeking a challenge. Threats come from extortionists, organized crime, educated and highly motivated criminals, and aggressive opponents trying to disrupt or even take control of services and capabilities. Rather than planning for random failure, we must assume that a single weak link can allow an entire system to be compromised.

### *SAMATE Project Background*

The NIST SAMATE project began in 2005 sponsored in part by the Department of Homeland Security. The goal was to develop methods, measures, and metrics to evaluate tools and techniques to determine how much they contribute to assurance. For instance, how much does the use of static source code analyzers help? How much assurance can we get from using test-driven development? More subtle questions ask whether the use of two techniques is additive or duplicative. For example, if a web interface is developed with the cleanroom approach, does running a web application scanner help enough to justify its use?

We in the SAMATE project identify tools, recruit focus groups experienced in the development and use of tools, develop testable behavioral specifications and test plans, and collect and write test material. The test material is publicly available, so all may benefit. By "tools" we mean a bundle of functionalities with a coherent purpose or approach. A single computer program may combine several conceptual tools for ease of use, speed, resource and result sharing, etc.

The project also organizes workshops and conference sessions to bring together researchers and users to foster collaboration, catalyze projects, and enhance communication.

As previously noted, SAMATE began with two tools or tool classes: source code security analysis tools and web application scanning tools. Both are automated analysis tools which can be used after production. We begin with the latter.

## Web Application Scanners

A web application is a program whose input and output is largely or primarily on the Internet, particularly the World Wide Web. Because of worldwide access, relative complexity, and rapid evolution, web applications are a spawning ground for vulnerabilities. A web application scanning tool, or web app scanner for short, heuristically tries dozens of different exploits and attacks on a web site and reports possible breaches it finds.

One challenge for web app scanners is that vulnerabilities are often not immediately apparent. The web app scanner must probe the web site looking for hints of vulnerabilities, slight weaknesses, or trivial malfunctions. When combined, these may yield an exploitable vulnerability. Yet, there are dozens of ways to probe a web site. How can one test a web app scanner that should use many, but definitely not all, of these ways to thoroughly probe a web site?

Another testing challenge is environmental complexity. SQL injection is a vulnerability in which the user can get SQL commands or command fragments through the web application front end into a database application. This allows the user to change or compromise data. To spur the web app scanner to exercise its SQL capabilities, the test environment must have most of the functionality of a database command handler and database tables. Another vulnerability, called cross-site scripting, occurs when a user can leave data on a web site, like a "comment", that compromises later web browsers attempting to render the "comment". So the test environment must supply much browser functionality, too.

A simple test approach of watching for particular probes quickly reaches a limit. No scanner needs to send all conceivable probes or even classes of probes. Yet, trying just one probe is clearly inadequate. A more severe limitation is that without feedback from the "web site" (test harness) that, say, a database is used, the scanner will not even try advanced probes to seek an SQL injection vulnerability.

A reasonable test approach is to simply build web applications with known vulnerabilities. Existing web servers, database products, and browser can be used to provide the infrastructure that web app scanners expect. Testers can examine the web app scanner report to determine if it finds and reports vulnerabilities. The OWASP WebGoat [2] is such an application, but with the purpose of teaching what to avoid.

One problem with this approach is the possible misuse of vulnerable applications. There is a chance that somebody bases production work on the test application. It may be tempting to start with the code and "fix" the weaknesses. Unfortunately it is hard enough to make an application secure when that is the goal from the outset. Starting with code having deliberately planted vulnerabilities cannot help assurance. Worse yet, some vulnerabilities allow applications or the machine on which they are running to be corrupted or crashed. Web app scanners try to avoid crashing the site being examined. But if a machine running a test application were exposed to the web at large, it could be compromised or corrupted.

### *Protocol for Researching Risky Software*

Clearly one must be careful when developing or using risky software, e.g. for testing. Risky software ranges from code with simple, known bugs to sophisticated worms and viruses with built-in polymorphic encoding and mutation engines to avoid detection. Such software may be root kits, which infect operating systems, or processes that contain many ways of invading and propagating to other hosts. It may be as innocuous as a bug that crashes the web server or database.

Working with risky agents is not unique to the computer community. Medical laboratories have four categories or levels of infectious agents defined [3], which parallel those in recombinant DNA research. The levels depend on the organism's potential to infect humans, the severity, and transmission vectors. For instance, "agents with a potential for respiratory transmission, and which may cause serious and potentially lethal infection" should be handled only at Biosafety Level 3 or above. Practices, safety equipment, and facilities are defined for each level. Also microorganisms may be engineered to be auxotrophs, that is, they cannot reproduce without some specific substance which does not occur in nature.

Computer hazards are not confined to vulnerable web applications developed to test. Anti-virus and anti-spyware companies work with code that might cause problems if it escapes. Protection agencies examine software that has deleterious effects on computer systems. Some colleges and universities assign students to write or work with viruses or malware to learn how to protect against it. At a minimum, software security trainees need practice with environments that have known vulnerabilities. Research labs for cell phone viruses, which have been demonstrated, or automobile Bluetooth links should be well isolated.

Computer science researchers working with risky software have a laudable record. Apparently adequate precautions are being taken. But new people are beginning such work. It would be helpful for them to have clear guidelines on what safeguards are prudent for what kinds of research with different kinds of risky software.

A web application, built for testing, with carefully introduced vulnerabilities might crash a server. Making the web application the computer analogy to an auxotroph may be sufficient. For instance, the application would refuse to run unless some file, like `enableRiskyWebApp`, is present or the system date is set to a specific year, like 2059. Other isolation possibilities are running on computers with no data connection to the outside (a so-called "air gap") or having network cards and USB storage device drivers removed.

We are planning to organize workshops to define levels or classes of risky software and develop protocols and recommendations for researching them.

## Source Code Security Analysis Tools

The second class of tool functionality the SAMATE project is working on is source code security analysis tools.

> "Source code security analysis tools scan a textual (human readable) version of source files that comprise a portion or all of an application program. These files may contain inadvertent or deliberate weaknesses that could lead to security vulnerability in the executable version of the application program." [4]

The work so far has produced a number or results. We released version 1.0 of a testable specification for such analyzers, have a draft test plan, and have written or collected several hundred programs and program snippets as test cases. In developing a specification for this tool class, we needed to address the different weaknesses that are, could, or should be caught. How could we have confidence that the assessment was thorough without a list of all (or all known) weaknesses?

In our August 2005 workshop, we brought up the need for a comprehensive, common list of software weaknesses and proposed a solution [5]. Since we had brought together many who had worked on taxonomies, this catalyzed the Common Weakness Enumeration (CWE) [6].

The specification for source code security analyzers consists of six mandatory features and three optional features, along with Annex A, a list of 21 weaknesses, and Annex B, a list of code complexities. Code complexities do not strictly affect the weakness, but do complicate analysis. Some complexities are loops, global or local variables, interprocedural calls, and indirection. Briefly, the requirements for mandatory features are

- Identify all weaknesses in Annex A.
- Report any weaknesses identified.
- Use a meaningful name for weaknesses.
- Give their directory, file, and line.
- Handle code complexities in Annex B.
- Have a low false positive rate.

We don't consider features dealing with ease of installation or use, integration with other tools, cost, or other important facets. Most of the requirements are straightforward, but why require a *low* false positive rate? Different uses have different acceptable rates. For a low criticality application just beginning to use a source code security analyzer, more than about 20 % false alarms may lead to rejection of the tool. On the other hand developers of a security-critical application may tolerate lots of false positive to minimize the chance of missing a real vulnerability. Since the requirement lacks a testable measure, why not remove it? NIST procedure is to only comment on requirements. Since the false positive rate is very important, we left the requirement. We plan studies and experiments to come up with a specific rate or rate classes for a future version.

The requirements for optional features are:

- Produce an XML-formatted report.
- Not report a suppressed weakness.
- Use the CWE name for weaknesses.

These are forward looking requirements. The XML formatting is for tool interoperability. Since there is no widely used standard, we don't specify the format to use. For greatest productivity, users need to be able to suppress the report of known weaknesses. They may be tolerated for many reasons: it is not really a weakness, something else in the system makes exploit of the weakness impossible, resources might better be used to fix other vulnerabilities, the risk is very low, etc.

When the CWE has matured, the requirement to use CWE names will replace the requirement to use meaningful names.

There are 175 test cases in C alone for the current draft test plan. C++ has about the same number, while Java has far fewer, since many common weaknesses cannot occur in Java at all. In addition to these test cases, we have an order of magnitude more examples that others and we have gathered or written for various investigations. For maximum flexibility and community benefit, we implemented a publicly accessible, on-line reference dataset to hold all these and more.

## SAMATE Reference Dataset (SRD)

The SAMATE Reference Dataset, or SRD, is a web-accessible database to search and share examples of code. There are currently more than

1800 entries of code in C, C++, Java, and PHP. Most of the entries have specific known weaknesses, although many have associated code with the weakness fixed. This corresponding "good" code is used to determine false positive rates. Each example has fields for meta-information, like author, submission date, a prose description, inputs to exploit the vulnerability, expected output, and compiler flags. The SRD also has test suites, which are defined sets of test cases.

The vast majority of the entries are small programs specifically written to illustrate some weakness. Some entries came from popular Internet applications, such as bind, sendmail, and wu-ftp, with known vulnerabilities [7]. Many other users, researchers, companies, and developers generously donated large and small sets of examples collected for various reasons.

The SRD is not limited to source code. The SRD can handle binaries and bytecode programs, which will be added to examine binary analyzers. We will also add examples in other languages, such as SPARK, Ada, and C#.

We planned the SRD to be a public resource. Users can search the test cases by language, description, author, weakness, status, strings in the source code, and many combinations. Registered users can comment on any test case, adding results of using certain tools or algorithms, observations on the legitimacy or limitations of the example, suggestions referring to related entries, etc. Approved people can add their own test cases or test suites to the SRD. Our goal is for the SRD to increasingly be a forum to exchange test material.

To serve as a foundation for research, the source code (or binaries, in the future) never change. Meta-information may be corrected, but the entries themselves are not fixed or modified. Thus when a researcher cites a certain suite of test cases, later developers can retrieve exactly the same test cases to see the improvement with a new algorithm. If an entry is determined to be bad, it can be marked as deprecated, and a replacement entry added. Deprecated entries should not be used for new work, and are not usually returned for searches, but they are still in the SRD.

Please contribute test cases your organization or you have developed to the SRD. Research and development of better tools and techniques could be sped up with a rich source of realistic examples. Although many people use the SRD, we are looking for comments and suggestions to improve it.

## Determining the Tool Efficacy

All this work leads to one primary question: how much do tools or techniques really help? In addition to developing standards and test materials, we have studies in progress. Dawson Engler asked, "Do software assurance tools really help improve security?" [8]. Certainly tools find weaknesses that can be, and are, fixed. But do tools find weaknesses that would be exploited if not fixed? Does fixing reported weaknesses introduce other, more subtle vulnerabilities? Perhaps the time teams spend checking what turns out to be false positives or fixing unimportant flaws could better be used in code reviews or design analysis. Rescorla studied whether searching for vulnerabilities increases security, but did not find clear evidence to support it [9]. Our preliminary studies of the effect of static analysis tools on software assurance have been similarly inconclusive [10].

As we said in the beginning, quality, security, safety, and other important properties must be designed in and built in from the beginning. We do not yet know how much tools and techniques contribute to assurance. There are many confounding factors to resolve, such as increased program use, misattributing vulnerabilities in other programs, program size, seasonal effects ("back to school"), and developer experience, to name a few. However we know that tools can give feedback on the development process and help developers learn what to avoid.

For further studies, we need your software development data: number of flaws found, type of weakness, number fixed, how found, etc. At NIST we are quite willing to operate under strict non-disclosure agreements. We have a reputation for handling the most sensitive of information.

## References

[1] The Software Assurance Metrics And Tool Evaluation (SAMATE) project, National Institute of Standards and Technology, http://samate.nist.gov/

[2] OWASP WebGoat Project, http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

[3] Biosafety in Microbiological and Biomedical Laboratories (BMBL), 1999, U.S. Department of Health and Human Services Centers for Disease Control and Prevention and National Institutes of Health, Fourth Edition. http://www.cdc.gov/OD/ohs/biosfty/bmbl4/bmbl4toc.htm

[4] Black, Paul E., Michael Kass, and Michael Koo, 2007, Source Code Security Analysis Tool Functional Specification Version 1.0, National Institute of Standards and Technology, Special Publication 500-268, page 4. http://samate.nist.gov/index.php/Source_Code_Security_Analysis

[5] Proceedings of Defining the State of the Art in Software Security Tools Workshop, Elizabeth Fong ed., National Institute of Standards and Technology, Special Publication 500-264, September 2005, pp. 79-82. http://samate.nist.gov/docs/NIST_Special_Publication_500-264.pdf

[6] Common Weakness Enumeration, MITRE, http://cwe.mitre.org/

[7] Zitser, Misha, Richard Lippmann, and Tim Leek, 2004, Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code, proc. 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-12), Newport Beach, CA, USA, ACM SIGSOFT, pp. 97-106. http://www.ll.mit.edu/IST/pubs/04_TestingStatic_Zitser.pdf

[8] Chou, Andy, Ben Chelf, Seth Hallem, Charles Henri-Gros, Bryan Fulton, Ted Unangst, Chris Zak, and Dawson Engler, 2005, Weird things that surprise academics trying to commercialize a static checking tool, invited talk at SPIN05 http://www.stanford.edu/~engler/spin05-coverity.pdf

[9] Rescorla, Eric, 2005, Is finding security holes a good idea?, Security & Privacy Magazine, IEEE, 3(1), pp. 14-19. http://www.rtfm.com/bugrate.pdf

[10] Okun, Vadim, William F. Guthrie, Romain Gaucher, and Paul E. Black, 2007, Effect of Static Analysis Tools on Software Security: Preliminary Investigation, Third Workshop on Quality of Protection (QoP'07), Alexandria, VA, to appear.

## Disclaimer

Certain trade names and company products are mentioned in the text or identified. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology (NIST), nor does it imply that the products are necessarily the best available for the purpose.

## Email Address

paul.black@nist.gov

*26th Digital Avionics Systems Conference*
*October 21, 2007*