SAMATE and Evaluating Static Analysis Tools

Paul E. Black

National Institute of Standards and Technology, 100 Bureau Drive Stop 8970, Gaithersburg, MD 20899; email: paul.black@nist.gov

Abstract

We give some background on the Software Assurance Metrics And Tool Evaluation (SAMATE) project and our decision to work on static source code security analyzers. We give our experience bringing government, vendors, and users together to develop a specification and tests to evaluate such analyzers. We also present preliminary results of our study on whether such tools reduce vulnerabilities in practice.

Keywords: software assurance, source code, static analysis, tool testing.

1 Introduction

The Software Assurance Metrics And Tool Evaluation, or SAMATE, project [11] at the US National Institute of Standards and Technology (NIST) focuses on one aspect of reliable software: software assurance, particularly security assurance. That is, how can we gain assurance that software is secure enough for its intended use? The SAMATE project seeks to help develop standard evaluation measures and methods for software assurance.

High levels of quality and security cannot be "tested into" software. Such attributes must be built into software from the beginning, starting with requirements and choice of environment. Preventing flaws is far more cost-effective and dependable than trying to remove them. 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? Legacy code may need reviews before being used in a new environment or for newly discovered threats. Also for quality assurance, one needs to know what kinds of flaws a current development process might leave or whether a new method yields better quality software. In all these cases, one must work with the code that is available.

Although SAMATE will eventually consider the impact of using better programming languages, such as Ada[1] or Eiffel, advanced software development approaches, and correct-by-construction techniques, we started with software metrics and understanding tools for checking software.

In the software realm, what can we do to increase software assurance? We can enable tool improvement and encourage wider use of tools. We will simultaneously urge use of better environments, practices, and languages.

[1] Any commercial product mentioned is for information only. It does not imply recommendation or endorsement by NIST nor does it imply that the products mentioned are necessarily the best available for the purpose.

Some questions quickly spring to mind. If a tool gives no outstanding alarms for a system, how secure is the system, really? Is the new version of a tool better (pick your own definition) than the preceding version? How much better? Which tools find what flaws? To answer these questions, we must back up and try to make a comprehensive list of flaws that might occur and have a taxonomy of software security assurance tools and techniques which might be investigated.

Both tasks have proved far harder than first thought. The effort to list flaws led to Mitre's Common Weakness Enumeration (CWE) [5] effort. Although not complete, the SAMATE web site has the latest version of the taxonomy of software security assurance tools and techniques [14].

For clarity we quote some definitions from our Source Code Security Analysis Tool Functional Specification Version 1.0 [13]. It defines a *vulnerability* as "a property of system security requirements, design, implementation, or operation that could be accidentally triggered or intentionally exploited and result in a security failure. … If there was a security failure, there must have been a vulnerability." It continues, "a vulnerability is the result of one or more *weaknesses* in requirements, design, implementation, or operation." We use the term "weakness" to emphasize that without the entire context, one cannot truly conclude that a problem may occur. Some other code or part of the system may prevent the weakness from being exploited.

Why we started with static source code analyzers

Higher level representations, such as requirements or use cases, are better places to prevent flaws. But we did not find any area mature enough for standardization. Also roughly half of all security weaknesses are introduced during coding [7], so improvement after high level design may be helpful. Unlike binary or byte code, source code is largely human-readable. Also there are many tools that work with source code. For these reasons, source code seems a good place to begin.

Testing and static analysis complement each other. Static analysis is less feasible without source code. It may be impossible, say, in testing embedded systems or remote testing of Internet services. On the other hand, one cannot test an incomplete program, while static analysis might be feasible. More importantly, testing is unlikely to uncover very special cases, for instance, granting access when the user name is "matahari".

We started with static source code security analysis tools since they have a potential of higher assurance. Chess and McGraw [3] give an excellent short introduction to the use of static analysis for software assurance. Note that when we use the term "tool", we are actually referring to a set of functionalities. That is, any program that can statically analyze source code is in the class. It need not be a stand-alone tool that analyzes source code and does nothing else.

Our first funding, from the US Department of Homeland Security, was to develop tests for tools. After talking with vendors, we decided we could increase adoption by developing basic tests.

One reason software developers do not adopt a tool is they are not sure whether any tool of the sort is really helpful and whether a particular tool is actually broadly useful. Some demonstrations are set up to give the appearance of great performance, when the breadth or depth or power is disappointing in practice. A standard developed by NIST would help assure software developers that source code security analyzers are useful. In addition, when a tool satisfies the standard, the user has some assurance that it has adequate coverage. Speeding the adoption of these tools can increase vendors' sales: the market for source code analyzers can grow a lot before it is saturated.

2 A specification for static source code security analyzers

NIST is a non-regulatory agency. This means we cannot mandate the use of our standards or tests. We must "sell" our results. One step to acceptable tests is a widely accepted specification of what such a tool should do.

We need the cooperation of vendors and developers of source code security analyzers to efficiently succeed. They have experience with what users, or at least their customers, need and want from such analyzers. They also have experience with what is practical, having written production analyzers. Some of the best researchers in the field have been the primary agents in developing these commercial tools. Finally we want our standard to be an endorsement that vendors will seek, rather than something to be forced on them. How can we position this effort to help vendors and consumers?

We looked for available source code security analyzers on the web, in published articles, and by personal contact. We the updated collection of tools is on-line at [12]. We organized several workshops and conference sessions on source code security analyzers. Vendors were willing to attend, and we discussed possible approaches and goals with them. To build a consensus, we established a mailing list, where we discussed facets, and made drafts of the specification available for public review and comment.

Informally a static source code security analyzer (1) examines source to (2) detect and report weaknesses that can lead to security vulnerabilities [13]. Tools that examine other artifacts, like requirements, byte code or binary, and tools that dynamically execute code are not included. Again, when we use the term "tool", we mean a set of capabilities of a tool.

Before continuing, we must decide the purpose of the specification and tests. It would be nice if they could serve as a metric to completely characterize the capabilities of a tool, but that is not possible, even in theory. A bit more practical specification could establish a lofty goal whose satisfaction ensures the user got the level of security checking needed. But since different users have different security needs, this is complicated. A specification could settle for a recommended standard, like due diligence. Even here, we have little objective evidence to establish such a level.

We chose to work for a minimum standard to begin with. A minimum standard would reduce argument about how high a level is right and exactly what should be required. Although insufficient for, say, setting recommended practices, a minimum standard opens the way for a higher standard.

What exactly should a source code analyzer do?

In more detail, such an analyzer should find weaknesses and report their class and location. The weakness class corresponds to CWE entries. Many tools also report conditions that may expose the weakness, data or control flow related to it, more information about that class of weakness including examples of how to fix it, the certainty that the weakness is a vulnerability (not a false alarm), or some rating of the severity or ease of exploit.

Optionally a tool should produce a report that could be used by other tools. For practical use in repeated runs, there must be some mechanism to suppress reports of weaknesses judged to be false alarms or otherwise to be subsequently ignored.

False positives are a critical factor. Conceptually, static analysis tools compute a model of a program. They then analyze the model for certain properties. Since static analysis problems are undecidable in general, either the computed model is approximate or the analysis is approximate. Due to these approximations, tools may miss weaknesses (false negatives) or report correct code as having a weakness (false positives). To be adopted a tool must "have an acceptably low false positive rate" [13].

Nowhere in the specification is a rate given. One reason is that a rate that is acceptable for one application or development situation may not be acceptable for another. Why then bother having the requirement? It is a NIST practice to only test items in the specification. It would be "poor sportsmanship" to test for a false positive rate without a written requirement. With more research we hope to be able to give acceptable rates, at least for some situations.

Other issues for a specification

The expressive power of programming languages makes analysis even harder. Analysis routines must be explicitly developed to handle coding complexities, such as loops, conditional control flow, arrays, different variable types, interprocedural function calls, and aliasing. In practice no tool handles all possible code constructs. To assure the user that a tool handles some, the specification also requires that weaknesses be found in the presence of a set of coding complexities.

No tool checks for all weaknesses in the CWE. Some are hard to define, like leftover debug code (CWE ID 489). With over 500 weaknesses tool developers concentrate on a relatively small set of frequent or severe weaknesses and put effort saved into improving analysis and user aids.

We chose a "minimum" set of weaknesses: those that are most common or occur most often, most easily exploited, and are caught by existing tools.

To be as flexible as possible the specification should explicitly refer to the subset of weaknesses that a tool purports to catch. However this causes severe problems in developing test material that also covers coding complexities.

We need to check that all weaknesses (in our "minimum" set) are caught in the presence of all coding complexities. The naive test suite would have every weakness in the presence of every coding complexity. This would be thousands of tests, complicating the creation and running of the test suite. Since we expect the same analysis modules to handle coding complexities for all weaknesses, we believe having each weakness in the presence of a few coding complexities, where every coding complexity occurs at least once, has substantially the same testing power. This test suite has less than 100 test cases.

Allowing for a subset of weaknesses presents challenges. In the extreme, what if a tool only purports to catch one weakness? The three or four test cases from the test suite will not exercise all coding complexities. We see a number of possibilities.

We could go back to having thousands of cases, so any weakness also exercises all coding complexities, but it would be unwieldy. One option is to develop a generator to create a custom test suite with the coding complexities distributed throughout as many or as few weaknesses as desired. Another possibility is to prepare adequate test suites as needed, in hopes that a limited amount of work would address real needs. Trusting that most tools cover a minimum set, we discarded the allowance for subsets from the specification. But we now find that unrealistic.

As part of the result of a study of tools Britton [2] reported, "84 percent of the vulnerabilities found were identified by one tool and one tool alone". Rutar, Almazan, and Foster [9] concluded that tools for finding bugs in Java do not overlap much in what they catch. In consolidating weakness classes found by five tools Martin [6] reported little overlap: few weaknesses were even caught by two tools. Currently the best approach, that is lowest false positive and highest identification rates over many weaknesses, is to use two or more tools as a combined metatool.

What are the attributes of test cases? Small cases separate the question of "can this be detected" from "how scalable is the tool". On the other hand, large programs allow examination of speed and maximum size and exercise a tool in a more realistic situation. Having one weakness in each test case makes analysis of the result easier, but having multiple weaknesses in one program should be more challenging. It is straightforward to write code with known weaknesses, whereas extracting examples from production code disarms the objection that it is unrealistic. Trying to find an instance of a weakness and extracting a slice of code could take excessive amounts of time. Even with the slice, we would have to secure permission to make it publicly available.

Currently our test cases are very small, purpose written code with one weakness per case. For measuring the false positive rate we also have cases without weaknesses or in which weaknesses have been fixed.

Sharing example code

While researching source code security analyzers, we found it difficult to get a corpus of code with known weaknesses. Although academicians develop them for research, companies have some for testing, and evaluators assemble them, few were available. We felt a single repository of such could be helpful to the entire community. Not only would it provide a place for us to keep and publish our test cases, it would allow people to share the work they've done.

The SAMATE Reference Dataset (SRD) [10] is an on-line, publicly available repository of thousands of samples of flawed software. Each test case consists of one or more files. Test cases may consist of source code, byte code, binaries, requirements, or other artifacts.

Each test case may 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. Registered users can submit test cases and add comments to any 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 new work. This way, a test report referring to a certain test suite can be rerun exactly, even years later.

Methods to minimize test evasion

A fixed, public test set allows for various abuses. A tool developer may write special-purpose code to get the right result for a very special case. This diverts effort from general improvements and incorrectly raises ratings. More simply a developer may add code to recognize the case, perhaps the name and size of the file, and hard-code the right result.

We see several ways to minimize such distractions. Easiest is to keep the test set secret. The test set would only be shared with parties trusted to keep it private. Practically a public version would be needed to allow others to examine and critique the tests and to allow vendors to practice. Even with a public version, it might be difficult to convince tool developers that the private tests are fair and reasonable.

Another possibility is to develop a test generator that creates unique test sets on demand. The challenge with this approach is to ensure that every test set generated is similar in its testing power. Adding code from production applications or getting large pieces of code would be hard.

We are writing an obfuscator to discourage developers from having their program recognize tests and return pre-determined answers. At a minimum the obfuscator must change source code file names. The next level is to change comments and names in the source code, such as variable and function names. Although very hard in general, the obfuscator only needs to work on the test suite.

Since most source code analyzers already have powerful abstraction capabilities, they could store signatures of abstract syntax trees and return prepared responses when one is recognized. To foil this, the obfuscator could insert benign code or rearrange existing code. Rather than requiring full code rewriting capabilities in the obfuscator, test cases could be in some preprocessed form or have "hints" stored. In this case, a macro processor could generate many different versions of the test set. Developers still might be tempted to add special case algorithms to improve results.

3 Do tools really help?

One can think of several potential problems with the use of such tools in practice. A tool may report many weaknesses, but miss the small number of serious weaknesses that really affect security. If a user takes a mechanical approach to fixing weaknesses reported by tools, programmers may not think as much about the program logic and miss more serious vulnerabilities. Also, the developer may spend time correcting unimportant weaknesses reported, making other mistakes in the process and not having as much time for harder security challenges. Recognizing such problems, Dawson Engler [4] articulated the question: "Do static source code analysis tools really help?"

Funded by the US Department of Homeland Security, Coverity, in collaboration with Stanford University, has analyzed over 50 open-source projects since March 2006 [1]. As an example, they reported over 600 defects in Firefox and 98 defects in Python. At least one security vulnerability was detected: CVE-2006-0745. Others have similar, although smaller, scans. Maintainers may use these reports to fix previously unknown vulnerabilities. By studying these, we may be able to support or refute Engler's question.

We are examining the history of reported vulnerabilities for the projects scanned by Coverity. We use reported vulnerabilities as a surrogate measure for actual vulnerabilities. The null hypothesis is that there is no change in the number of reported vulnerabilities after the start of scanning. We give preliminary results we have for one project, MySQL.

Coverity scanned MySQL version 4.1.8 in early 2005. Version 4.1.10, released 15 February 2005, contained fixes based on Coverity reports. Figure 1 compares vulnerabilities discovered in version 4.1.10 or later versions with vulnerabilities discovered before the 15 February release. "Discovery" means it was reported in the National Vulnerability Database (NVD) [8].

Red bars, on the right, are vulnerabilities discovered in version 4.1.10 or later. They are grouped by discovery date. As the discovery date, we used the earlier of the discovery date in the NVD and in the SecurityFocus database [15]. Our data covers 21 months after the release of version 4.1.10. The light blue bars, on the left, are vulnerabilities discovered before the release. We began 21 months before the release, that is May 2003. Vulnerabilities discovered after 15 February 2005 that were only present in versions before 4.1.10 were not used.

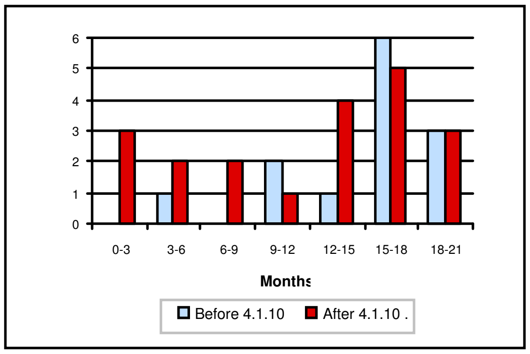

Figure 1 MySQL vulnerabilities before and after 4.1.10

The data is insufficient to draw any conclusions. We are trying to take confounding factors into account, and we are analyzing similar data from other projects to accumulate a statistically meaningful set.

4 Future directions

We are planning several studies to answer questions such as the following. How does one tool's assessment correlate with another tool's assessment? What is the subject of a metric, that is, does it apply to the algorithm, an implementation, or an execution trace?

We are currently working on specifications and tests for web application scanners. The next class of tool we will work on is binary analyzer. We are also guiding efforts to formalize descriptions of weaknesses. Although formal description will have many uses in the long term, our immediate application is a test case generator. The generator uses the descriptions to produce example code.

We are looking for collaborations. In particular, we need a few more people to serve on our technical advisory panel, which meets once or twice a year to suggest where we might be of most help in the future. We also seek participants in focus groups to review specifications and test plans for classes of tools.

In the long term we plan to go beyond tools, especially checking tools. Society must move beyond a catch-and-patch approach. We will help develop metrics to gauge more secure languages, good processes, environments, etc. We want to help demonstrate what really improves software security.

References

[1] *Accelerating Open Source Quality*, http://scan.coverity.com/ (Accessed 21 May 2007).

[2] Peter A. Buxbaum (2007), *All for one, but not one for all*, Government Computer News, 26(6), 19 March. Available at http://www.gcn.com/print/26_06/43320-1.html (Accessed 22 May 2007).

[3] Brian Chess and Gary McGraw (2004), *Static Analysis for Security*, Security and Privacy Magazine, IEEE, 2(6), pp 76-79.

[4] Andy Chou, Ben Chelf, Seth Hallem, Charles Henri-Gros, Bryan Fulton, Ted Unangst, Chris Zak, Dawson Engler, *Weird things that surprise academics trying to commercialize a static checking tool*, http://www.stanford.edu/~engler/spin05-coverity.pdf (Accessed 21 May 2007).

[5] *Common Weakness Enumeration*, The MITRE Corporation, http://cwe.mitre.org/ (Accessed 21 May 2007).

[6] Robert A. Martin (2007), *Making Security Measurable*, Providing Assurance in the Software Lifecycle DHS-DoD Software Assurance Forum, Fair Lakes, Virginia.

[7] Gary McGraw (2006), *Software Security*, Addison-Wesley.

[8] *National Vulnerability Database*, National Institute of Standards and Technology, http://nvd.nist.gov/ (Accessed 21 May 2007).

[9] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster (2004), *A Comparison of Bug Finding Tools for Java*, 15th IEEE International Symposium on Software Reliability Engineering, IEEE Computer Society, pp 245-256. Available at http://www.cs.umd.edu/ ~jfoster/papers/issre04.pdf (Accessed 21 May 2007).

[10] *SAMATE Reference Dataset*, National Institute of Standards and Technology, http://samate.nist.gov/ SRD/ (Accessed 20 May 2007).

[11] *Software Assurance Metrics And Tool Evaluation (SAMATE) project*, National Institute of Standards and Technology, http://samate.nist.gov/ (Accessed 20 May 2007).

[12] *Source Code Security Analysis*, National Institute of Standards and Technology, http://samate.nist.gov/ index.php/Source_Code_Security_Analysis (Accessed 24 May 2007).

[13] *Source Code Security Analysis Tool Functional Specification Version 1.0*, National Institute of Standards and Technology, Special Publication 500-268, May 2007. Available at http://samate.nist.gov/ docs/source_code_security_analysis_spec_SP500-268.pdf (Accessed 24 May 2007).

[14] *Tool Taxonomy*, National Institute of Standards and Technology, http://samate.nist.gov/index.php/ Tool_Taxonomy (Accessed 25 May 2007).

[15] *Vulnerabilities*, SecurityFocus, http://www.security focus.com/vulnerabilities (Accessed 25 May 2007).